1

AGARD-CP-439

AGARD CONFERENCE PROCEEDINGS No.439

Software Engineering and its Application to Avionics

DISTRIBUTION AND AVAILABILITY
ON BACK COVER

AGARD-CP-439

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARD Conference Proceedings No.439

SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS

Papers presented at the Avionics Panel Symposium held in Ceşme, Turkey, 25—29 April 1988.

The content of this publication has been reproduced directly from material supplied by AGARD or the authors.

Published November 1988

Copyright © AGARD 1988
All Rights Reserved

ISBN 92-835-0483-6

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

THEME

Software Engineering has evolved rapidly but the gap between demand and software output continues to grow. By the end of the decade research programmes in North America, Europe and Japan will begin to produce results in the areas of software tools, computer architecture, etc. The symposium considered how these advances might be applied to the avionics systems of the nineties and beyond and their impact on our aspirations in areas such as operation, fault tolerance etc. The software element of modern weapon systems continues to grow in size and complexity; offering major advantages but also potential risks.

(NATO · FURNISHED) (RH)

L'ingénierie du logiciel a été l'objet d'une évolution rapide, mais le fossé qui existe entre la demande et la production de logiciels ne cesse de s'élargir. A la fin de la décennie, les programmes de recherche lancés en Amérique du Nord, en Europe et au Japon commenceront à donner des résultats dans le domaine des outils-logiciels, de l'architecture des ordinateurs, etc. Le symposium a étudié la façon dont ces progrès pourraient être appliqués aux systèmes d'avionique des années 90 et au-delà ainsi que leur impact sur nos aspirations dans des domaines tels que la facilité de mise en oeuvre, l'insensibilité aux défaillances, etc. L'élément logiciel des systèmes d'armes modernes ne cesse de grandir en taille et en complexité, en présentant des avantages importants, mais aussi des risques potentiels.

AVIONICS PANEL

TECHNICAL PROGRAMME COMMITTEE

PROGRAMME COMMITTEE MEMBERS

AVIONICS PANEL EXECUTIVE

CONTENTS

*Not available at time of printing

Page

TECHNICAL SUMMARY
55th MEETING OF THE AVIONICS PANEL
ON
"SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS"

by
D.V.Gaggin
Director, US Army Avionics R & D Activities
Attn. SAVAA-DD
Fort Monmouth N.J. 07703-5401
United States

The symposium was judged by the panel members to be extremely successful. The large majority of papers were of a high quality, the attendance was excellent throughout the meeting and the audience participation was exceptionally good. Several key points were continuously brought out and are worth highlighting here.

The development process can be represented as a water wheel whereby the user needs become the system requirements which become software requirements which are implemented in code, etc. until a final product is delivered back to the user. This process takes many years and involves hundreds of people even for the simplest systems. Any misunderstanding along the way is carried through to the end item. It is imperative that the key individuals in the chain have direct access to the user and completely understand the original user's need.

Ada will be the standard higher order language of the near future. The Ada development/support environment is expanding rapidly but needs maturity before it receives wide acceptance. Ada will not solve all higher order language problems and will continue to evolve as more people use and become familiar with it. But, like any standard, it is a snapshot in time and will be replaced by far more advanced systems such as GRAPH. These new languages are not in conflict with Ada but a natural technology evolution.

Documentation is a major concern. It is expensive to generate, time consuming and since it is by definition one of the last functions to be completed in the development phase it is one of the easiest things to eliminate when a program runs into funding difficulties. But since documentation is critical during the operation and support phase, we must place greater importance on it during development.

Expert systems are beginning to emerge and appear to be extremely useful in coping with our increasingly complex systems. Initial application will probably be concentrated in ground support systems and ILS tools before maturing into prime mission equipment. True artificial intelligence is still in the research stage with no near term applications foreseen.

Advances in software technology are being exceeded by the rate of growth in avionic application. Major efforts are mandatory to improve software technology. For example, innovative techniques are required to permit reusable software; integrated tools are needed to allow automatic software production in order to increase productivity.

Software reliability has become an important issue in mission safety critical avionic systems. Fault tolerance techniques and methodologies are being developed to address this issue.

Integrated Programming Support Environments are not now available but are badly needed and slowly being developed.

Design methods still tend to be non-standard but Ada is pushing more and more designers towards an object oriented approach.

Finally, requirements definitions which historically have used natural language appear to be tending towards more formal approaches such as structured analysis. Even though these methods are more rigorous this will be slow in coming due to reduced readability.

ADA THE SOFTWARE ENGINEERING TOOL FOR AVIONICS KEYNOTE ADDRESS

by

J.Batz
ADA Joint Program Office
Rm 3E11, The Pentagon
Washington D.C. 20301
USA

The text of this address was not available at the time of printing.

DISCUSSION

W.Mansel, Ge

1. Missile software is divided into two parts; guidance software located within the missile and management software located on the aircraft. Does CAMP address both parts?
2. Is CAMP available outside US?

Author's Reply

1. CAMP software is grouped into general and non-general classifications. It is my understanding that the general parts perform support functions which are applicable to a wide variety of real-time applications, whereas the "non-general" parts are associated specifically with missile avionics functions which are common among many missile systems. Intuitively, I believe that the latter could include functions like navigation, guidance and perhaps power management. But it is less clear that detailed functions associated with flight control can be made common, since each missile system will have its own unique set of control surfaces, propulsion systems and attitude control thrusters, etc. I will try to provide more detailed information on the actual parts being developed under the CAMP project.
2. I am hopeful that at least a reasonable level of detail on the CAMP software parts can be provided. I am uncertain what level of information can be provided outside the US and therefore do not know if the Ada source code can be provided.

D.Gaggin, US

Many people are concerned about using Ada on large software development projects because of the lack of software tool maturity. What is the track record of such programs as far as keeping within cost and schedule predictions?

Author's Reply

I do not have any information on track records for programs employing Ada. However, the maturity (and quality) of the software tools available is strictly a function of the particular compiler and support system vendor chosen. That choice is inherently related to the target processors to be employed in the real-time system. The choice, therefore, must be made in a coordinated way. If the target processor choice is made without regard to the compilers and tools available to support software implementation for the target processors, then the program managers are choosing a high risk approach. The Ada Compiler Evaluation Capability (ACEC) sponsored by the AJPO is an effort to help program managers make wise choices in this area.

J.Lacroix, Fr

What improvements in Ada real-time performance do you expect from the new tools?

Author's Reply

I do not see new tools as improving Ada *real-time performance*, but as reducing the time and cost for development and maintenance.

The improvement in real-time performance will be a function of several other actions: (1) The improvement of compilers to produce more efficient target code, (2) Improvement of run-time operating systems and (3) The improvement of target processors for real-time control operation. It would be best for these actions to be taken in a highly coordinated manner, but they will be done largely by different sets of people for different motivations. Furthermore, the actions needed may differ from compiler to compiler, from machine to machine and run-time operating system to run-time operating system. In addition, modest changes to the Ada language itself may come about in the next few years to enable a better coupling between Ada constructs design explicity for real-time applications and machine characteristics needed for efficient real-time control operation.

Measures of Merit for Advanced Military Avionics: A Users' Perspective on Software Utility

Major Mark C. Dickerson, Experimental Test Pilot, F-16 Combined Test Force
Mr. Victor M. LaSaxon, Manager-Aircraft Flight Test Engineering (ETSS)
Computer Sciences Corporation

6510th Test Wing
Edwards AFB, California 93523-5000

SUMMARY

This paper is intended to provide recommendations for improving the software development process. The recommendations are the result of discussions with F-15, F-16, and HH-60 test pilots, navigators, and flight test engineers. Because today's aircraft are becoming so software intensive, and because the development process is so involved, users and developers must redouble efforts to design systems correctly the first time.

Although 29 specific recommendations are given, they can be boiled down to four general guidelines:

A. Keep switch actions to a minimum.
B. Keep switchology consistent.
C. Tailor displays by flight phase, but keep other options visible.
D. MOST IMPORTANT, carefully process and meter the information presented to avoid pilot overload.

NOTE: Annotations of the form (R17) appear throughout the text. They refer to specific recommendations listed in the last section of the paper. In this case (recommendation 17.)

TODAY'S SYSTEMS

Today's aircraft rely heavily on software intensive advanced avionics systems (See figure 1). Indeed, the importance of this is evidenced by the fact that many modern fighters undergo planned, deliberate software and hardware evolutionary steps. The F-15 and F-18 multistaged improvement programs (MSIP) are examples of this. Specifically, McDonnell Douglas Corporation's Pace Eagle and Advanced Hornet programs are seeking to define future changes in the F-15 and F-18 so as to keep the aircraft operationally competitive against the evolving threat. These programs have sought to group upgrades in an orderly and planned fashion.

The F-18 attack capability improvement will be one area that will encompass a great deal of advanced avionics software. This software will be designed so as to use and integrate the functions of new equipment such as the Flight Incident Recorder and Monitoring System (FIRAMS), a built-in test and health monitoring system that is an improvement on the current maintenance signal data recording system; the AIM-120 advanced medium range air-to-air missile (AMRAAM), an airborne self-protection jammer and, of course, the AGM-65F air-to-surface imaging infrared Maverick missile. Other planned improvements that will require a large software role is the addition of a Hughes forward-looking infrared (FLIR) navigation system for enhanced night attack capability, raster head-up (HUD) display and night vision goggle compatibility.

Possible additions to the F-15 include new weapons systems, electronically scanned antennas for the radar and new sensing systems such as infrared sensors. The F-15E program is equally as active in the advanced avionics software arena. McDonnell Douglas is adding night, all-weather strike/attack capability to the already advanced F-15 Eagle. It will have double the central computer capability in the same size box as that of the first production MSIP aircraft. The APG-70 radar capability in the F-15E has a 512-kilobyte capacity. This is a significant increase from the APG-63 384K capacity.

The General Dynamics F-16 multinational staged improvement program goes back to 1979, and like the F-15 and F-18 programs, it is very dependent on advanced avionics software. Block 40 aircraft will have a primary mission role of night, low altitude precision strike under adverse weather conditions. Computer Sciences Corp. is aiding the Air Force in integration and flight test of all F-16 MSIP avionics including the Martin Marietta low altitude navigation and targeting infrared for night (LANTIRN) pod system as well as the Rockwell Collins global positioning system (GPS) for very precise position, velocity and time information. The Block 40 aircraft will also contain a number of specific software enhancements so as to improve pilot-vehicle interface. A Teledyne general avionics computer (GAC) will replace the present fire control computer. This computer contains four times the memory and three times the computational capability of the current F-16C/D versions. Radio frequency compatibility will be totally software programmable on Block 40 aircraft with the implementation of an advanced interference blanker unit. It will accommodate up to six matrices of actions to be taken.

Within the program, General Dynamics Ft. Worth Division is also working on the configuration and avionics of the Block 50 aircraft. These changes resulted from a joint General Dynamics Defense Department emerging threats study. Upgrades will be designed to improve performance in the air-to-air arena, to add more advanced night and day awareness and threat warning systems to counter the Soviets' look-down, shoot-down radar capabilities, and to continue efforts to make the pilot's job easier and more efficient.

TOMORROW'S SYSTEMS

The current crop of avionics software available to the aviator provides a virtual avalanche of "data" or "facts" suitable for analysis and interpretation. The next generation of pilot aids will advance from this "data" oriented software to an "information" network displayed to the pilot at the appropriate time or available on demand (see figure 2). We are currently witnessing the evolutionary process taking place. For instance, although altitude is displayed on the HUD, some fighter aircraft have the added feature of a "break X" low altitude cue as well as an audible "PULL UP" command from the computerized voice generator. The critical difference between data or fact oriented software and more "information oriented" software is the interpretive element of the algorithms. The future software package will be designed to help the pilot maintain timely and effective situational assessment. The system will act to a great degree as an extension of his senses and perceptions. It will also provide sufficient automation to manage certain subsystems. This will allow the pilot to maintain the "big picture" and not be distracted by mundane simple tasks. This should result in improved pilot response time, particularly during heavily tasked, high stress and/or high threat situations. Integration of sensors, flight control, fire control, countermeasures and weapons with a beyond-visual-range capability will provide the pilot situational awareness capabilities designed to increase his ability to assess, accurately monitor and quickly react to the tactical situation. Furthermore, with this advanced software, the system will be able to incorporate tactical information from airborne or land based integrated battle management command posts.

Several software types will be used to accomplish the goal of subsystems management, enhanced beyond visual range capability, and situational assessment, and display. These include three dimensional color graphics drivers, speech recognition and synthesis routines, multisensor image recognition, basic system software, as well as software for weapons delivery. The situation assessment software will be designed to integrate multiple sensor inputs from onboard as well as from data linked sources if available. Once enough information has been assembled, these elements may even provide the pilot with beyond-visual-range identification information. The system would then establish the magnitudes and immediacy of threats, storing the data in memory and alerting the pilot. Programs will focus on the integrated attack scenario and will display engagement options that also include maintaining combat flight integrity. The ultimate objective is to optimize attack opportunities for the flight while minimizing risk exposure.

As the information is accumulated, it will be prioritized and displayed in a manner so as not to overload or distract the pilot. One concept being developed would use a miniature, helmet-mounted television display. Avionics software would command symbology showing the location of both airborne and land based threats. Additionally, the display would indicate weapon status and modes as well as the position of friendly forces. The concept also includes a sophisticated array of sounds and verbal cues that could alert the pilot to threats at his six o'clock position as well as indicating his own flight conditions. All of these informational displays will be flexible and fully programmable by type mission and pilot preference. The pilot will be able to see any sensor information superimposed over the entire field of view or a selected portion. In addition, the pilot will have infrared imaging, a large menu of navigational aids and the ability to magnify an area of particular interest.

Mode selections and commands will be designed so as to reduce pilot workload. Flight gloves may be equipped with magnetic location devices so as to allow the pilot to request information by pointing his finger. Also, there may be a limited ability to employ voice commands.

THE ACQUSITION PROCESS

As might be expected, the process for acquiring new software systems like these is usually lengthy. Figure 3 shows the major steps in the USAF acquisition chain. This process can take in excess of three years for a software design change on a mature system. It is even longer for a new system. The portions of this process where the software engineer has the greatest opportunity to influence the results are in the design and design review phases. Unfortunately, pilot input to the product traditionally occurs as part of the Statement of Need (long before the engineer begins a design), at the design reviews (by which time the design is virtually complete), and during flight test (which is a very expensive way to find problems).

The problems with this should be obvious. There are several levels between the initial pilot inputs and the initial design effort. Close scrutiny will reveal that although some of the people involved at these levels are engineers, few are likely to be pilots. In fact, most of them will be in marketing, contracting, or public relations. These people may have no clue what the pilot wanted, nor what kind of software effort will be needed to provide it. They know neither the software engineer's job, nor the pilot's, yet they are the channel through which the pilots' requests must flow in order to get to the software engineer.

The designer and the pilot must make an effort at the start of design work to talk face to face and work shoulder to shoulder. They must be sure that what is being developed is what the user really requested. (R18) They must also work--sometimes at odds--to be sure that what users have requested is what they really need. To this end, Northrop Corp used pilot "murder boards" during F-20 software development. These groups of experienced pilots would discuss employment concepts, hammer out a unified position, and submit it directly to the software engineers, with virtually no middlemen involved. The Air Force test pilot felt the results were excellent.

The Users' Groups and Cockpit Review Teams which are listed in the figure deserve elaboration. These groups normally consist of a handful of highly experienced, highly respected pilots. These individuals are delegated the authority to speak for large numbers of pilots in their respective services or countries, and they usually have sufficient backing to make their opinions stick. These users will discuss potential mechanizations, evaluate contractor proposals, and vote on what they feel will be the best approach to a given problem. Normally the results of these votes will form the basis for a contract to modify a current software package, or to design a new one.

Often members of the Users' Group will attend the preliminary or critical design reviews to confirm that the system looks the way they expected it to look. Having a simulation available for pilot evaluation, even a rudimentary one, can do a lot to confirm the accuracy of the mechanization before the more costly flight testing phase. (R-19) When these simulations are used, the actual code that will support the aircraft itself should be used if at all possible. This not only saves work for the programmer, but it gives the pilot a chance to see the "real thing" to the maximum extent possible. (R-20) It may also give the software designer the leverage to push for earlier hardware development so that it can be evaluated along with the software.

When simulation is performed, every effort should be made to design some quantitative form of evaluation into the simulation. Many decisions are made simply on the basis of what a certain pilot "likes" or "dislikes". In these cases, the more forceful or authoritative pilots may tend to sway other evaluators based solely on strength of personality. If very specific tasks are laid out--the CORRECT tasks--then there will be little room for argument. Being able to write that "Mechanization-B had a 30% lower average time required for accomplishing the task," makes the results difficult to argue with. (R-21)

F-15 PILOT COMMENTS

The F-15 is a two-engine, all-weather fighter-interceptor built by McDonnell-Douglas. Originally designed primarily for the air-to-air role, the aircraft has subsequently been modified to perform the all-weather long-range interdiction mission as well (F-15E). It has also been selected as a technology demonstrator for advanced short takeoff and landing (STOL) concepts.

F-15 pilots and weapon system officers (WSOs) familiar with all of these variants were asked to evaluate how effectively the avionics and software support their missions. The evaluators were all members of the F-15 Combined Test Force at Edwards AFB, California. To simplify discussion, we have broken the mission down into the Pre-takeoff, Cruise, Combat, and Recovery phases. Surprisingly, a large proportion of aircrew comments related to the Pre-takeoff phase. This is probably because that is the phase in which the most interaction with onboard computers takes place--data is going from the crew to the aircraft. Keep in mind that most, if not all, problem areas discussed here will have been corrected by the time the systems are fielded. Our objective is to avoid tripping over the same problems.

In any advanced multirole aircraft like the F-15E (figure 4), a lot of keyboard work is required to set up all the systems. It should be no surprise that pilots prefer automated data entry during the Pre-takeoff phase. (R2) During the remainder of the flight, data goes both ways, but the majority of the information flow is from the aircraft to the crew, so displays tend to dominate the discussion.

PRE-TAKEOFF: Although menu-driven software is generally preferred, some testers felt that the weapons loading menus in the F-15 could be improved. Pilots reported that it was unnecessarily difficult to change from a branch entered in error and get back to the correct branch. They felt that a single action "Home" button would allow them to quickly abort and restart a transaction when loading weapons. (R1) They also noted that two different menus had very similar names, making it even more likely that an error might occur. Specifically, one page was labeled "ARMNT", the other was "WPN". The ARMNT page was the page required to load stores into inventory, but because of similar names, it was easy to inadvertently select the WPN page instead. The WPN page was actually video select for an electro-optical weapon. What also annoyed them was that this option appeared even if no such weapon was loaded. The computer was "knowingly" presenting an option tht didn't exist. (R3, R4)

One pilot felt that there were actually several unnecessary branches. There was one branch for loading air-to-ground weapons, another for air-to-air, and yet another for "simulated" ordnance. This pilot made the point that the aircraft has only so many stations. Why should he need to change loading modes and work down through another decision tree when all he really wants to do is load another station? Also, instead of a special "training" branch, why not just allow loading of zero quantity? This could flag it as a simulated load and eliminate the need for a separate branch. Another pilot noted that some options required a "select" followed by "enter", others required only a "select". A consistent technique would be better. (R7)

Although loading modes showed room for improvement, pilots praised the ability to pre-load several different weapon delivery programs. The ability to quickly select the desired program then allowed rapid reaction to pop-up targets once in combat, and the correct delivery mode was only one switch away. One pilot suggested that the same ability should exist for stores jettison modes, so that he could quickly select any of several combinations of ordnance and/or tanks to leave the aircraft. (R12)

In the F-15E, an up front control panel was added to make systems easier to access. Although the idea itself was appreciated, some crewmembers felt that the mechanization required too many pages of submenu to reach the desired systems. Regularly used systems should have easily accessible menu pages. (R17)

One last comment on Pre-takeoff procedures involved the radar. It seems that the system would always power up in a certain mode, no matter what mode the pilot may have selected on the control panel. This would require that the mode be recycled to achieve the requested configuration. It isn't unreasonable for a system to require a certain mode in the early or warm-up phase; however, when the warm-up is complete the system should align itself with what the user has requested. On an air-to-air alert scramble, possibly air base defense, the pilot should be able to set up his cockpit so that no more than two switch actions are required to put a missile in the air. He shouldn't have to waste two actions just getting a system to do what he has already directed it to do. (R9)

CRUISE/COMBAT: Although it is admittedly a hardware oriented item, a mandatory requirement for any interdiction aircraft is an intelligent autopilot. (R5) Specifically, the INS is programmed with route information. The autopilot can fly the aircraft. Both systems are on the aircraft data bus. There is no reason why the aircraft shouldn't be able to proceed autonomously to the target. Hidden in this requirement is the need for high reliability in any software which is coupling into the flight controls. At low level, in the weather, there is little margin for error when the aircraft begins its own turn to the next point on the route.

A common thread among several pilots was the concern for information management. We have aircraft with multiple sensors and integrated avionics. These sensors have ability to acquire much more data than ever before. Unfortunately, human input ports and maximum throughput rates have changed little. We must be selective in what we choose to display, and be sure that it is displayed clearly. For example, aircrew were very enthusiastic about the Tactical Situation Display. It provides a moving map showing own ship position, as well as locations of airborne targets acquired by various aircraft sensors. The "God's Eye" view was considered easy to use and interpret.

On the other hand, some crewmembers were disappointed in the use of new multicolor displays in the weapons management mode. The capabilities seemed to be underutilized. Instead of drawing in weapons and using the various colors to highlight weapon status, the same old monochromatic abbreviations were used as before. This forced the pilot to come inside, focus, read, and interpret--wasting time. (R6) Although these displays have limitations relating to glare, color washout, and resolution, they do have advantages upon which we can capitalize. (R8)

Pilots were also disappointed with a complex F-15A radar display. The radar attempted to present so much data about each target that it became too cluttered to use against large formations. The symbols were large, each extending almost 1/4 of the length of the screen. When targets began to maneuver, one pilot described the result as looking like a sword fight. This display may have been a result of the tendency to make "evolutionary" or incremental improvements to systems and displays without considering their effect on the system as a whole. Usually it is easier to add code to an operational flight program (OFP) than it is to improve the code that is already there. This is the beginning of information overload. (R13)

As users and developers, we must be smart about how much information we display, and about what is presented in the heat of battle. For example, one pilot commented that during the final stages of an engagement, his primary concern is whether or not he has reached valid shot parameters. He felt that a "shoot" cue should stand out from everything else, even if your head is out of the cockpit. Current "shoot cues" may actually be a step backward from those employed in later models of the F-4, simply because they are so small. (R10) This small size may be a direct result of trying to put too much data on the available displays, or it may be because the displays are not properly "missionized" to take into account pilot workload as a function of phase of flight.

One way to take the onus off of the designer regarding what and how much to present is to make displays more "tailorable". If the pilot, via automated data transfer equipment, can personally tailor his displays prior to flight as a function of flight mode, he can find the combination which works best for his own mission and experience level. This "tailorability" doesn't have to be unlimited. Just offering a choice of several optional chunks of information can go a long way toward making a single OFP do the work of several.

(R12) In the same vein, the pilot shouldn't be denied access to information which is readily available to the computer. One pilot complained that certain radar failure states are accessible on the ground, but are unavailable once he is airborne. Nothing is more annoying than to be denied information by your own computer. (R11)

RECOVERY: Pilots were very impressed with the Klopstein format landing display planned for the STOL demonstrator. In this display, the runway is represented by a trapezoidal figure displayed in the HUD. This figure has the same relative size and shape as the runway would if it were visible. (See figure 5.) A picture is still worth a thousand words. (R6)

HH-60 PILOT COMMENTS

The HH-60A (Night Hawk) is a twin-turbine engine, single-rotor, semimonocoque fuselage, rotary wing aircraft, built by Sikorsky. It is designed for search and rescue. With two flight deck crew it is able to carry up to ten passengers and a crew plus internal and external cargo. The cockpit (figure 6), designed by IBM, is highly integrated so that the aircraft can be operated without a flight engineer. Features include a three-cue flight director function, an alert by exception philosophy, and computer monitoring of normally static engine and airframe instruments. It also has automatic initialization of most avionic control functions with default values that, once altered during the mission, remain that way. (R14) The HH-60 also has a flight planning software function, fuel and power management computer calculations and integrated basic flight instrumentation on multipurpose displays (MPD).

We solicited pilot comments from individuals familiar with HH-60 flight test results. In general, they concurred that the avionics software as designed was suitable for the mission. They were complementary about the numerous options available to the pilot although there were some minor descrepancies with the choice and consistency of some of the nomenclature. (R7) The two major complaints concerning the avionics were the fact that they felt overwhelmed with information and that the system definitely was not user friendly. Since there was so much data available there were many layers of software the crew had to travel through to get to what was desired. A colorful example that illustrates this concerned the fifteen steps it took to set the altimeter which could then only be displayed by going three pages deep into the software. The initialization, recovery, and system status monitoring (IR&SSM) function is the backbone of the avionics system. It is designed to perform initialization and monitoring of all avionics subsystems as well as redundancy control of the avionics and pilot interfaces. System initialization was satisfactory as demonstrated in 196 flights. This allowed the operator to power up the avionics and begin normal operations generally without manual initialization of any avionics subsystem. The pilots would have liked the system embellished with the capability to cartridge load mission specific information and parameters. They felt that the quick response search and rescue role would be enhanced by freeing the pilots from having to load all of the information manually. (R2)

Avionics subsystem power transient recovery was rated as marginal. In particular, the system status monitoring function displayed a wide range of false failures after power transients which leads to some additional pilot comments concerning software system design. (R9) Even without the added variables introduced due to the power transients, the approach to system status monitoring left something to be desired in the pilots opinion. As stated earlier, the system uses an alert by exception philosophy. This of course means that you must rely on the judgment and proper functioning of the monitoring system to know when something goes wrong. There are undesireable aspects to this philosophy. First, the pilot has no continuous way to monitor certain instruments to establish trends so as to anticipate problems. (R15) A simple case in point is the fuel display. It takes a deliberate act of judgment on the part of the pilot for him to display fuel status. This of course means that if this is not done on a regular basis, it is possible for the crew not to realize that the external tanks did not feed until they get the fuel low light. Thus the crew could find itself deep into enemy territory or far over the water without enough usable fuel to make it back. Furthermore, if the accuracy of certain displays is effected by power transients, the crew would not have any baseline data upon which to develop manual estimates since they had not been continuously monitoring the systems.

The flight information display is divided into five basic categories: flight instrument values; engine and airframe instrument values; avionics flight symbology; warning/cautions/advisories (WCA); and general display control. Discussion of this area led to very useful comments concerning display philosophy vis-a-vis software implementation. Chief among these comments was the need for what one pilot called a "scratch pad". The general function of this display is to be a catch all of inputted data to be used for monitoring or crew coordination. Case in point was the IFF squawk on the HH-60. The pilots complained that it took too many switch actions on their part to see what they were currently squawking. The same was true of the comm frequency. They felt that these could be displayed on the "scratch pad". Also, this display could be used when a new input was being made on a given system to avoid confusion, duplication of effort and aid crew coordination. (R16)

The second comment concerning displays was the use of analog versus digital formats in the software. This would allow the pilot to use trend information such as acceleration or decceleration, which is more easily accounted for with analog type displays. (R15) A display format featuring simulated "round dials" for critical instruments could make trend information available at the pilots' option.

Next, the pilots expressed their preference for color displays with appropriate symbology. Also, an HH-60 feature they enjoyed was the ability to declutter the display panel for certain phases of their mission. The last major comment concerning displays was their desire for the ability to mode select without releasing flight controls. (R29)

In summary, avionics software concerns revolved around information management, consistent programming and nomenclature, user friendly selection process, and nonlabor intensive power-up and data loading processes.

F-16 PILOT COMMENTS

The F-16 is a single-seat, single-engine, multirole fighter made by General Dynamics. It was designed as a lightweight fighter and optimized for the day, clear weather dogfight role. It also has very good all-weather surface attack capability. The Falcon is flown by nations as diverse as Pakistan, Isreal, and the United States, as well as several European countries. It was one of the first aircraft to employ the MIL-STD 1553 data bus to tie multiple components together into an integrated avionics suite. The two major variants are the F-16A/B and the F-16C/D. From a pilot perspective the major difference between the two versions is the use of multifunction displays (MFDs) in the C/D variant for increased system flexibility. Typical cockpit layouts are shown in figures 7 and 8.

Characteristics of the avionics were discussed with USAF test pilots who had experience in both variants of the aircraft. Some comments mirrored those of F-15 pilots covered earlier, but since the F-16 started off as a computer-oriented machine, many problem areas associated with pre-takeoff operations have already been addressed. Therefore pilots tend to have more comments on combat related capabilities.

PRE-TAKEOFF: Pilots generally praised the display and stores management mechanizations. A common rule throughout the system is "Press to change". If what the pilot sees is not what he wants, he simply presses the label. This will rotary through other available modes or provide a menu of them. The general feeling was that if there were three or fewer options, a rotary was fine. For four or more options, a menu seemed easier to use. (R24) Refer to figure 9 for examples of the process.

As with the F-15, stores loading drew several comments. For the most part, the A/B stores loading mechanization was preferred. This is a menu-driven load procedure which lists stores by abbreviations of their actual names. The pilot selects the stations which he'd like to load, and the stores management system (SMS) shows the options available. The pilot then pages through the catalog and picks out the desired store. In the C/D mechanization, no options are presented. The pilot has to select the desired station(s), then key in a code for the desired store. This code is a three-digit number having no logical connection with the store itself, and no list of options is presented. The pilot must simply know the code. If this type mechanization is to be used, one pilot suggested that a "Help" page be added. This would cross reference the weapons with the associated codes. (R25,R26) In addition, both mechanizations drew criticism because racks and pylons had to be loaded separately, even if the store loaded could only be carried on a specific rack--extra work for no reason. (R8)

One feature that pilots felt was very useful was the ability to tailor certain air-to-air modes prior to flight. These modes allowed the pilot to use a single switch action inflight to set up radar search pattern as well as weapon choice. (R12) The ability to set up a master mode (air-to-ground (A-G) for example) exit that mode for air-to-air (A-A) if needed, and return to A-G without having to reprogram it was also considered valuable. (R27)

CRUISE/COMBAT: Every pilot expressed concern about the amount of head-down work required to use the fire control and navigation panel (FCNP) in the A/B model. This particular panel must be accessed in order to select certain radar modes, some weapons delivery modes, and to update the INS. Although this is a hardware-oriented problem, one possible solution would be to put two or three of the most heavily used functions into hands-on or up-front controls and then display the information momentarily on the HUD. (R29) The data should be deselectable, because some pilots felt that the HUD was already becoming cluttered, particularly in C/D variants. (R10) Many numbers are presented in the HUD and most are unlabeled. As a result, the display has become very difficult to interpret at a glance.

Regarding the HUD, pilots appreciatd the use of analog type displays for altitude and airspeed. (R15) One also praised the commonality between A-A and A-G "in range" displays. The HUD flashes whenever entering weapon delivery parameters. The message is the same: "time to press the button". The wide angle raster (WAR) HUDs, in combination with LANTIRN pods, have made night or adverse weather navigation easier, but they also spread out the altitude and airspeed information. One pilot noted that the visual angle between altitude and airspeed in the WAR HUD is actually larger than the angle between the two gages on the instrument panel!

Radar displays and modes also generated some comments. In general, pilots were pleased with the radar presentations in both models of the F-16. Although very different, both are effective. Several felt that hands-on control of azimuth scan in a manner similar to the F-15 would be desireable. Others wanted a heads-up way to change the scan patterns. For example, the radar can search from side-to-side at a constant elevation, or it can change elevation slightly from sweep-to-sweep and search a larger volume. These pilots wanted to be able to change the pattern without having to go head-down to the radar panel. The radar was also tied into the voice message unit (VMU), so that in certain radar automatic search and lock modes, the pilot would get a "lock" call in his headset. This would allow a pilot approaching an area with known hostiles to maintain simultaneous visual and radar search. (R8) A final comment regarded the A/B radar display, which presents a block of information about the target in a section of the display. In some instances, the target would drift into the information block and be occluded. A hands-on declutter solved the problem by removing the data block on request.

Several comments regarding different systems emphasized a pilot desire for consistent switchology and displays. One button on the side-stick has multiple uses. As the nose-gear steering switch, it toggles the steering on and off. As the missile step button, it rotaries through all loaded missiles of a given type. But during certain A-G deliveries, it becomes a one-way switch to go from the currently selected mode to a continuously computed impact point (CCIP) delivery mode. Pressing the switch again has no effect. A switch which has always been a rotary has suddenly gone bad, and the pilot must go back to the SMS to regain the previous delivery mode. (R7)

On the positive side, pilots were generally pleased with the target management switch (TMS) on the stick. No matter what mode the pilot had selected, the TMS always seemed to do what was expected: it always caused a sensor or system to become stabilized wherever it happened to be looking or located at the time. No thinking was required, because the purpose was so consistent from mode to mode. (R23)

RECOVERY: One software feature given high marks by pilots was a set of "cruise" modes. These are essentially fuel management modes which give the pilot speed and/or altitude cues for best using his remaining fuel. "Endurance", "Range", and "Home" modes would not only provide the proper speed and altitudes to fly as a function of gross weight, but they would also take into account the drag of all loaded stores in making the computations. The "Home" mode would also give the pilot a predicted fuel remaining when overhead the field. Software like this makes superb use of the things a computer does best. It provides "information" to the pilot rather than unprocessed "data". (R8)

RECOMMENDATIONS

The specific recommendations below arise as a direct result of user interviews discussed earlier. Each is directly related to one or more specific suggestions. They are not listed in any particular order, but the authors recommend special consideration be given, as a minimum, to four underlying concepts:

A. Keep switch actions to a minimum, particularly in the heat of battle or for error recovery.
B. Keep switchology consistant throughout the mechanization.
C. Keep "display options" visible, but tailor default displays specially for the planned phase of flight.
D. MOST IMPORTANT, provide the pilot with useable "information". Megabytes of unprocessed "data" only increase pilot workload. PRE-PROCESS IT!

Specific Recommendations

1. Always include a single action "return to start" capability in any menu-driven software. (This is in addition to the standard "back-up on page" capability.) Single action recovery from erroneous keystrokes is also a must.

2. Automate data entry whenever feasible. (This is obviously a hardware intensive recommendation.)

3. Don't display "non-options". If certain software paths are deadends, don't allow the pilot to enter them.

4. Avoid using similar words that could generate confusion.

5. To reduce workload, get the proper systems talking to each other, rather than to the pilot. INS should talk to flight controls, RWR should talk to ECM, etc.

6. Use pictures instead of words whenever possible. If color is available, use it to maximum advantage.

7. Be consistent, not only in mode selection switchology, but in displays as well.

8. Use the computational capabilities to improve a mechanization rather than "putting the old mechanization on the computer".

9. Systems should eventually reach the modes commanded by the current switch placement. If intermediate modes are required, the system should cycle through them automatically, even if the recycle is due to a power transient.

10. "Missionize" the displays to emphasize the correct information and reduce clutter, but make additional information available if specifically requested.

11. The system shouldn't keep secrets. The pilots should be able to read (but not necessarily change) any mission or failure state information in the computer if he has the time to access it.

12. Pre-programming combat modes prior to flight can save precious seconds later.

13. Be disciplined about adding "bells and whistles". Don't just present more data because it's available. Process the "data" and present "information". Be sure the addition will be useable in the heat of battle.

14. Smart initialization is a must for any quick reaction system. They should be set up for the "most probable" quick response scenario, not the "worst case".

15. Long-term trend information on systems like engine and fuel should be easy to access. Short-term trend information for things like altitude or airspeed must be constantly displayed, preferable in analog format. For these applications, standard mechanical displays may be better than digital.

16. For multiplace aircraft, a scratch pad type display might improve information flow throughout the aircraft.

17. Often changed items like radio frequency, altimeter and squawk shouldn't be hidden behind menu pages. They should be readily available at all times.

18. There must be increased pilot participation early in the software design phase to make certain that what users' have requested is understood by the actual programmers.

19. Simulations, even simple PC-based ones, should be used whenever possible to show pilots how planned mechanizations will work.

20. When simulations are used, every effort should be made to use the actual hardware and software that will be used in flight.

21. Quantitative measures of performance should be used during pilot evaluations if possible. This will produce more objective results.

22. Aural cues are a good option for warning information. When they are used, words are better than more horns or tones. For normal trends (like airspeed) artificial wind noise might be appropriate.

23. A specific switch should do functionally equivalent things no matter what master mode the system is in.

24. Be flexible regarding rotary versus menu formats for switches. When options are few, use a rotary. When options are many, use a menu.

25. Menu-driven software is generally preferable to code word driven, both for mental workload as well as reduced switch actions.

26. Make "Help" pages available for complex or code word oriented sections of the mechanization.

27. Weapons modes should not reinitialize when exited and re-entered. They should remain as the pilot last saw them.

28. Whenever possible, avoid the use of simultaneous switch actions (such as designate and hold while slewing).

29. Use the hands on throttle and stick (HOTAS) philosophy for heat of battle switch actions.

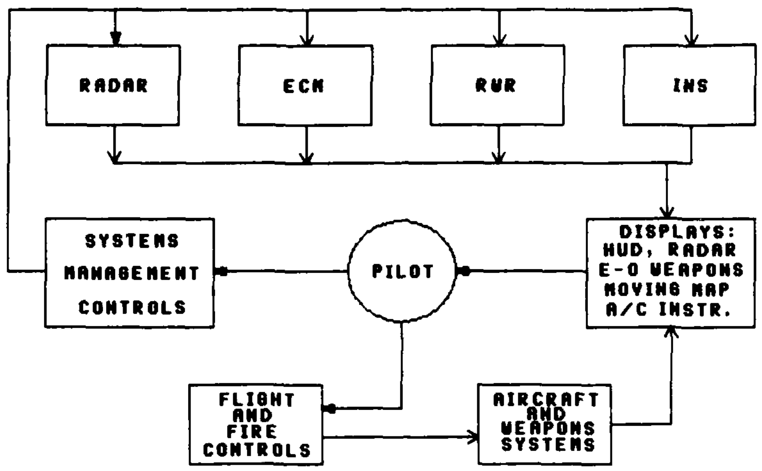

RADAR
ECM
RWR
INS
SYSTEMS MANAGEMENT CONTROLS
PILOT
DISPLAYS: HUD, RADAR E-O WEAPONS MOVING MAP A/C INSTR.
FLIGHT AND FIRE CONTROLS
AIRCRAFT AND WEAPONS SYSTEMS

Figure 1
TODAY'S ADVANCED AVIONICS SYSTEMS

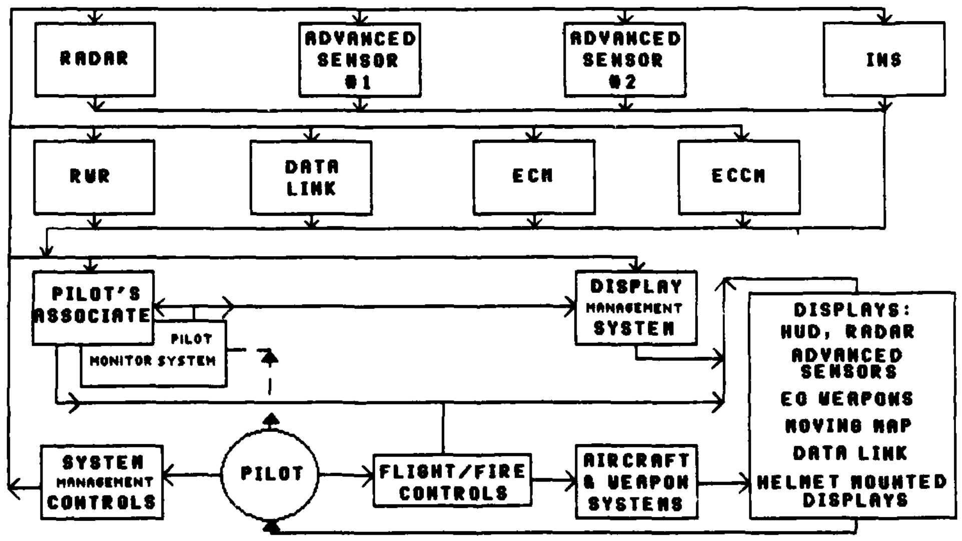

RADAR
ADVANCED SENSOR #1
ADVANCED SENSOR #2
INS
RWR
DATA LINK
ECM
ECCM
PILOT'S ASSOCIATE
PILOT MONITOR SYSTEM
DISPLAY MANAGEMENT SYSTEM
DISPLAYS: HUD, RADAR ADVANCED SENSORS EO WEAPONS MOVING MAP DATA LINK HELMET MOUNTED DISPLAYS
SYSTEM MANAGEMENT CONTROLS
PILOT
FLIGHT/FIRE CONTROLS
AIRCRAFT & WEAPON SYSTEMS

Figure 2
TOMORROW'S AVIONICS SYSTEMS

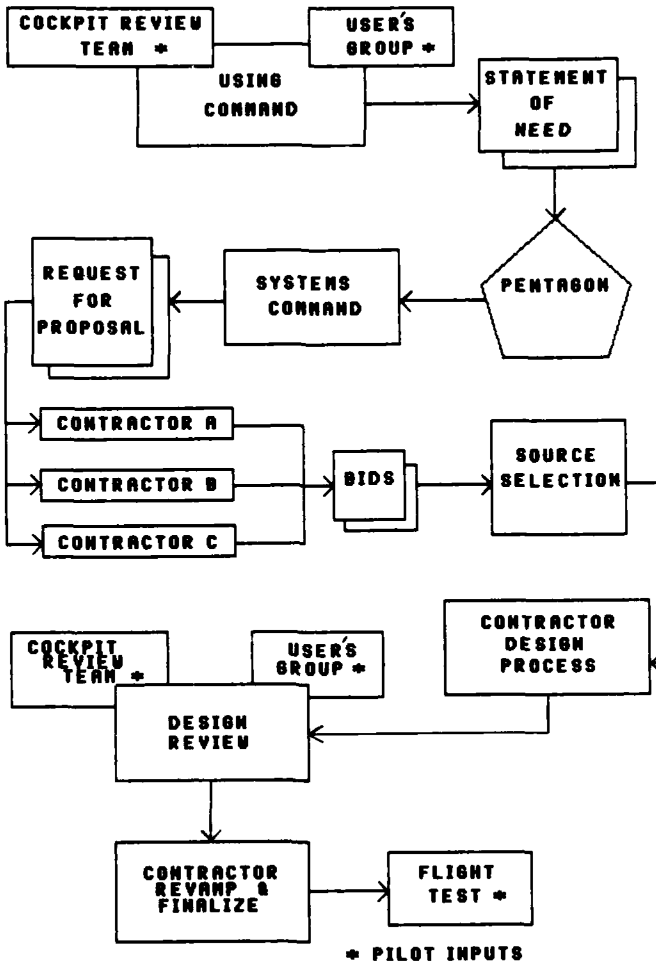


Figure 3
USAF SYSTEMS ACQUISITION
(abbreviated)

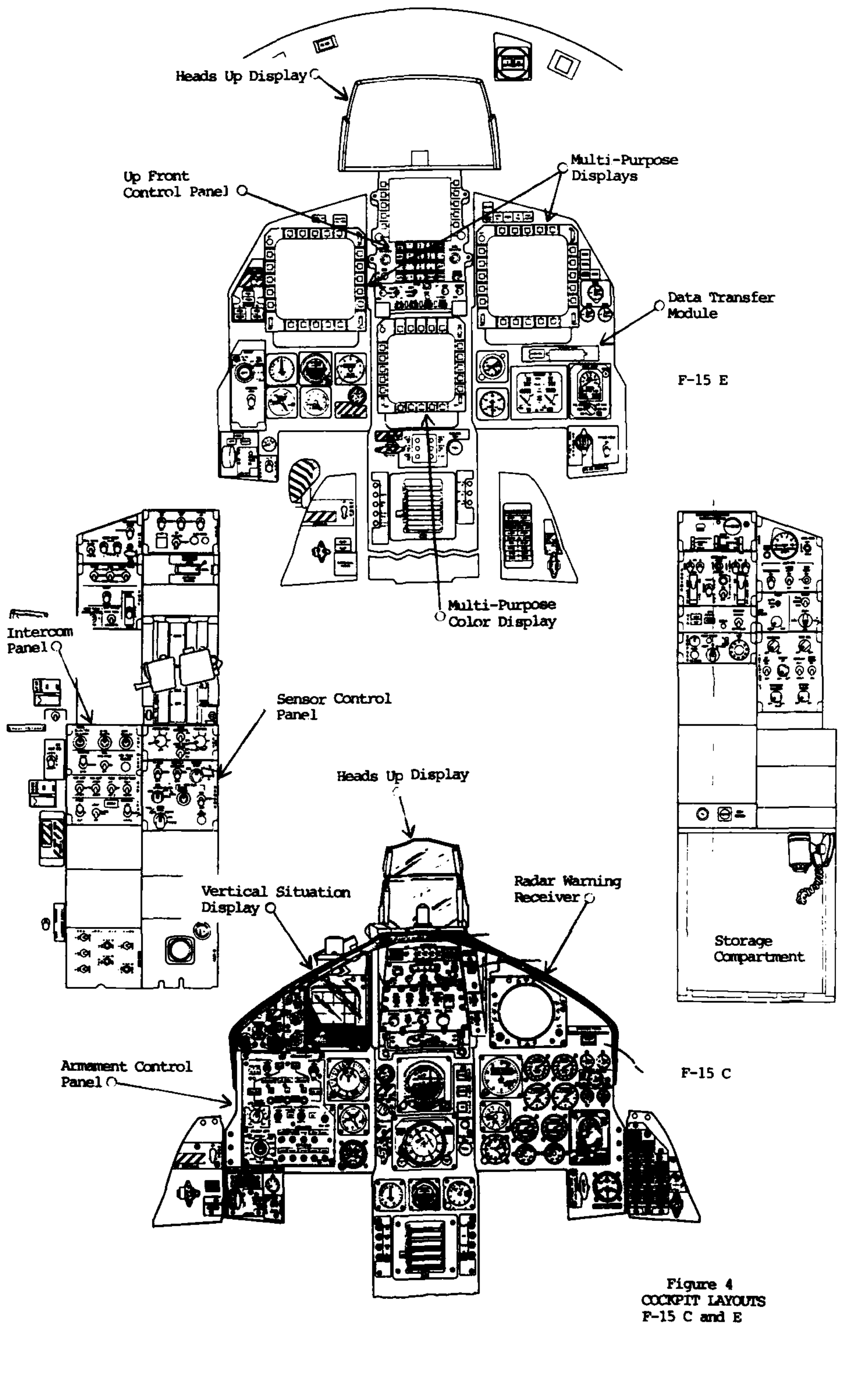
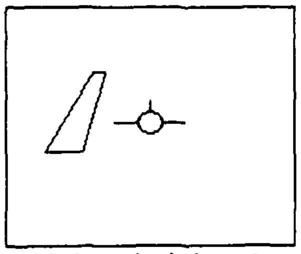
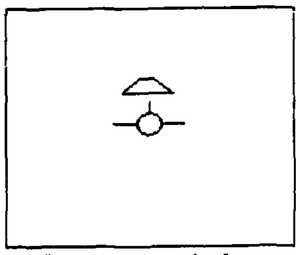


Figure 4
COCKPIT LAYOUTS
F-15 C and E



High and right of course



On course, below glide path

Figure 5
F-15 STOL Klopstein Format
HUD Landing Display.

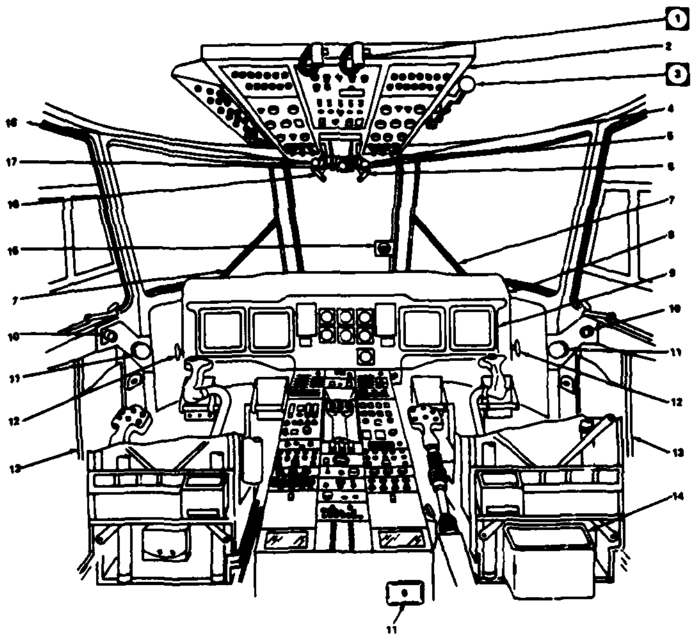
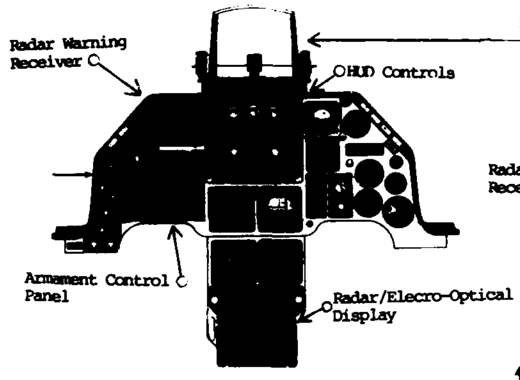
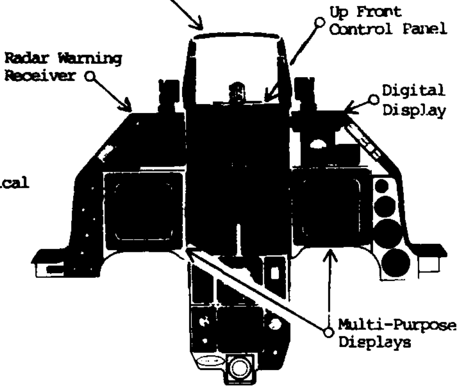


Figure 6
HH-60 COCKPIT LAYOUT

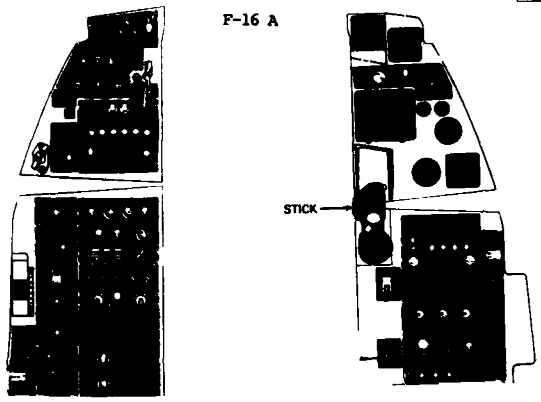
1. COCKPIT UTILITY LIGHTS	10. VENT/DEFOGGER
2. UPPER CONSOLE	11. ASHTRAY
3. ROTOR BRAKE	12. PEDAL ADJUST LEVER
4. NO. 2 ENGINE FUEL SELECTOR LEVER	13. MAP/DATA CASE
5. NO. 2 ENGINE OFF/FIRE T-HANDLE	14. STORAGE BIN
6. NO. 2 ENGINE POWER CONTROL LEVER	15. STANDBY (MAGNETIC) COMPASS
7. WINDSHIELD WIPER	16. NO. 1 ENGINE POWER CONTROL LEVER
8. INSTRUMENT PANEL GLARE SHIELD	17. NO. 1 ENGINE OFF/FIRE T-HANDLE
9. INSTRUMENT PANEL	18. NO. 1 ENGINE FUEL SELECTOR LEVER



F-16 A



F-16 C



Figures 7 and 8
F-16 A and C COCKPIT LAYOUTS

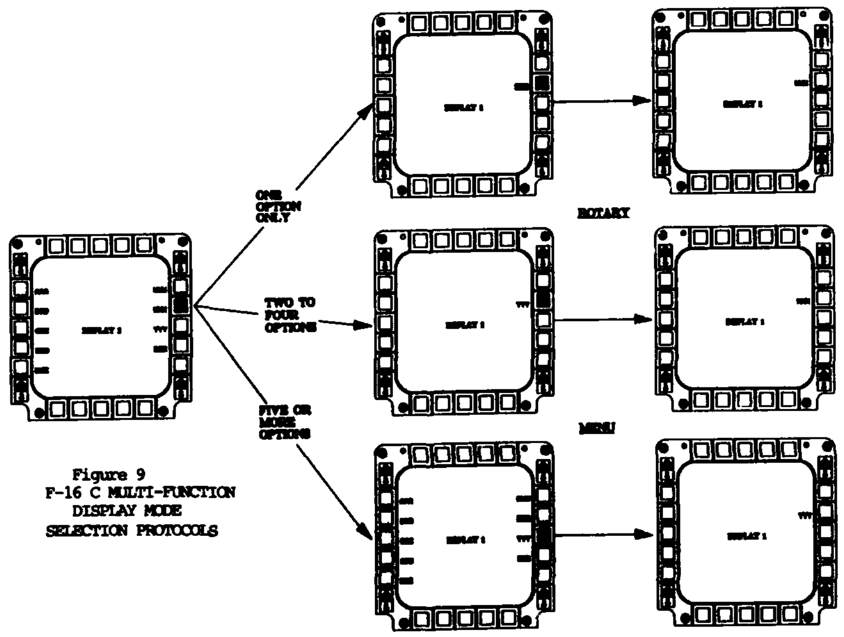


Figure 9
F-16 C MULTI-FUNCTION DISPLAY MODE SELECTION PROTOCOLS

BIBLIOGRAPHY

1. Christiansen, et. al., Too Much, Too Soon: Information Overload, IEEE Spectrum, June 1987.

2. Crowley, Software Acquisition and Management, USAF Acquisition Management School, Brooks AFB, Texas, November 1987.

3. Duffy, et. al., Automation in Combat Aircraft, National Research Council, Washington DC, 1982.

4. Fink, et. al., Fighter Modernization, Aviation Week and Space Technology, 21 July 1986.

5. Fowler, Comments on Cost and Peformance of Military Systems, IEEE Transactions on Aero and Electronic Systems, January 1979.

6. Greenstein and Revesman, Simulation Studies of Human - Computer Communication, IEEE Transactions on Systems, Man and Cybernetics, October 1986.

7. Morishige, Cockpit Automation - A Pilot's Perspective, AGARD Joint Guidance, Control, and Flight Mechanics Panel; Stuttgart, West Germany, October 1987.

8. Munson, F-16 ADA Simulation Plan, General Dynamics, Ft. Worth, Texas, March 1987.

9. Newbauer et. al., Digital Avionics, Aerospace America, December 1986.

10. Royer, F-16 ADA Simulation Report, General Dynamics, Ft. Worth, Texas, April 1987.

11. Skira, Integrating the Future Fighter's Flight Controls, Aerospace America, July 1984.

12. Smith and Mosier, Guidelines for Designing User Interface Software, MITRE Corp, Bedford, Massachusetts.

13. USAF Test Pilot School, Human Factors in Systems Testing, Edwards AFB, California, June 1983.

14. USAF Test Pilot School, Integrated Avionics Systems Testing Theory, Edwards AFB, California, September 1985.

DISCUSSION

W.Urschel, US

Current design efforts have used one of two extremes — either the pilots are ignored by the designs until flight test or the pilots are brought in early via cockpit working groups and end up judging or proposing specific design solutions, not just functional requirements. What is the solution to this?

Author's Reply

Pilot evaluation of proposed changes through simulation facilities allows decisions on adoption of changes to be based on empirical test results, not just biased opinions. Thus, each proposed mechanization or change should be evaluated by multiple pilots on the simulator and decisions on implementation should be based on simulation evaluation results, not just individual pilot's desires.

To summarize:

a. Get pilot inputs early.
b. Validate plans via realistic operational tasks.
c. Keep pilots involved as system takes shape.

J.G.Brevot, Fr

1. What is the schedule for the completion of the "Pilot's Associate" (PA) program?
2. What are the first sophisticated applications that you envision in this program?

Author's Reply

1. Since the development of the PA concept will be evolutionary, incorporating improvements and modifications gradually, no actual "completion" is likely...possibly ever. Like an air-to-air missile program, new and better versions will continue to be tested and built.
2. This is probably the more important question, also more difficult. I would expect some forms of safety-oriented functions, like automatic terrain avoidance, automatic aircraft out-of-control recovery or automatic air-to-ground weapons delivery to be the most reasonable systems to be operational first. There may also be better target recognition systems within the next few years.

LE RESPONSABLE DE L'EXPRESSION DU BESOIN FACE AUX INCERTITUDES LIEES AUX TECHNIQUES DE L'INTELLIGENCE ARTIFICIELLE ET DES SYSTEMES EXPERTS SUR LES AVIONS DE COMBAT

par

Colonel Jean-Georges BREVOT
Chef de la Division Nouveaux avions de combat
Bureau des programmes de matériels
Etat-major de l'armée de l'air française
24, boulevard Victor - 75996 PARIS ARMEES
FRANCE

SOMMAIRE

Cette communication analyse d'une part le travail que doit mener le responsable de l'expression du besoin pour rédiger les spécifications d'un avion de combat, et d'autre part les questions qu'il se pose sur les possibilités réelles offertes par les techniques de l'intelligence artificielle et des systèmes experts.

L'auteur en déduit que la spécification de l'emploi de ces techniques sur avion de combat constitue une sérieuse difficulté et propose des solutions pour surmonter cette difficulté.

ABREVIATIONS

IA	Intelligence artificielle
SE	Système expert

L'apparition des techniques de l'intelligence artificielle dans les domaines de l'aéronautique militaire est un fait marquant de ces dernières années.

Le responsable de l'expression du besoin, dont le travail consiste à prévoir les conditions d'exécution future des missions, se doit d'en tenir compte. Il se heurte cependant à un certain nombre d'incertitudes liées à ces nouvelles techniques, incertitudes venant s'ajouter à celles qui accompagnent normalement le travail de rédaction des spécifications opérationnelles.

Nous allons analyser les incertitudes de l'expression du besoin, celles liées à l'intelligence artificielle et tenter de dégager quelques solutions pour exploiter au mieux les possibilités nouvelles.

1 - L'EXPRESSION DU BESOIN

1.1 - Les spécifications opérationnelles

Le travail du responsable de l'expression du besoin en Etat-major se heurte à de nombreuses difficultés, en particulier lorsqu'il a trait au domaine des avions de combat.

Il consiste en effet à savoir se projeter dans l'avenir, et le cheminement de la pensée permettant de dégager les caractéristiques essentielles du besoin doit en fait tenir compte de nombreuses incertitudes.

Une incertitude importante tient au fait que l'adversaire ne nous donne pas ses intentions et il convient de tenter de les prévoir. Une autre incertitude est due aux percées technologiques souvent imprévisibles.

Toutes les réflexions doivent prendre place dans un cadre représenté par la menace. Or cette menace dépend bien entendu de la technologie dont dispose l'adversaire, mais également de l'évolution des situations géopolitiques et des rapports de force. Chacun sait que cette évolution est particulièrement difficile à prévoir.

Tout ce travail se traduit par la rédaction des spécifications opérationnelles qui constituent un document de base auquel se réfère constamment l'utilisateur au cours du déroulement d'un programme.

Si ce document est bien rédigé :

- on peut escompter que le matériel remplira le besoin, c'est-à-dire qu'il se montrera efficace pour remplir la mission,
- mais on peut également escompter que le matériel remplira juste le besoin ; ce qui signifie que son coût sera raisonnable c'est-à-dire rentrant dans le volume financier qui peut être consacré au programme.

1.2 - La prévision de la mission

Il s'agit donc de prévoir la nature et les conditions d'exécution de la mission dans un futur à long terme et d'en déduire le besoin correspondant.

Mais de quel long terme s'agit-il ? En moyenne le développement d'un avion de combat prend 10 ans, et ensuite l'avion va avoir une durée de vie dans l'Armée de l'air utilisatrice de l'ordre de 25 ans, *avec très souvent une rénovation à mi-vie. Il faut donc prévoir* l'évolution *des stratégies* et des tactiques sur une période de 35 ans, ce qui n'est pas une tâche aisée.

Voici à titre d'exemple quelques unes des grandes questions qui se posent dans la prévision de ce futur à long terme.

- Les missions offensives seront-elles exécutées par avions isolés, par patrouille ou par raids massifs ?
- Compte tenu des menaces et des parades possibles, les altitudes optimales seront-elles *basses ou hautes* ?

  Quelles seront les vitesses associées ?
- Faudra-t-il privilégier la vitesse ou la manoeuvrabilité ?
- Les missions de supériorité aérienne s'effectueront-elles sous guidage complet ou au contraire de façon la plus autonome possible ?
- Quelle sera la qualité de l'identification et de la coordination sur le champ de bataille ?

Mais quelles que soient les solutions envisagées, il reste que les matériels devront toujours être conçus pour permettre d'exploiter au mieux les qualités immuables de l'aviation de combat : la souplesse d'emploi, l'aptitude à la concentration, la rapidité d'action dans la profondeur.

1.3 - L'évolution technologique

Une des tâches essentielles du responsable de l'expression du besoin consiste à suivre avec attention l'évolution des technologies et à en prévoir les tendances. Malheureusement son travail est compliqué par le fait que cette évolution n'est jamais une progression constante. L'émergence de nouvelles technologies peut créer des percées difficilement prévisibles contre lesquelles il convient de se prémunir, ou au contraire qu'il faut tenter d'exploiter.

Cependant on peut constater en cette matière une caractéristique qui s'est toujours vérifiée dans l'histoire des techniques militaires : l'alternance entre l'avantage de l'épée et celui de la cuirasse. Actuellement on peut constater un avantage assez net en faveur de l'épée, c'est-à-dire de l'action offensive, due à la sophistication des systèmes d'armes, des armements et des missiles. Mais les tendances technologiques actuelles, en particulier dans les domaines des contre-mesures, de la furtivité ou du durcissement des objectifs, font penser qu'on pourrait assister à un retour de l'avantage en faveur de la cuirasse c'est-à-dire de la position défensive.

Voici à titre d'exemple quelques domaines technologiques ayant eu ou devant avoir un impact certain sur les conditions d'exécution des missions, donc sur la définition du besoin en matière d'avions de combat :

- les progrès dans l'efficacité des missiles,
- l'arrivée des commandes de vol électriques,
- la mise au point des techniques pulse-doppler sur les radars de bord,
- l'optronique,
- l'augmentation de la précision de navigation,
- les possibilités de coordination du champ de bataille,
- l'augmentation explosive de la puissance des calculateurs de bord,
- et bien entendu l'arrivée des techniques de l'intelligence artificielle et des systèmes experts.

1.4 - Les dilemnes

1.4.1 - Le dilemne efficacité / coûts - délais

La première qualité d'un matériel militaire est d'être efficace. Mais cette efficacité n'est pas forcément obtenue par l'accumulation de fonctions. Elle est plutôt le résultat de la création d'un ensemble cohérent.

Cependant la réalisation de cet ensemble cohérent se heurte aux limitations dues aux coûts et aux délais.

- La limitation essentielle due aux coûts est celle qui ressort du débat qualité / quantité dans un volume budgétaire donné. L'impact du facteur quantité ne doit pas être sous-estimé car il apporte d'une part l'effet de nombre en matière d'efficacité opérationnelle et d'autre part l'effet de série entraînant une réduction des prix unitaires.

- La limitation due aux délais vient ensuite et oblige à savoir arrêter la définition du matériel sur une technologie donnée et à éviter le perfectionnisme.

1.4.2 - Le dilemne demande excessive / exploitation des possibilités

La limitation essentielle que rencontre donc le responsable de l'expression du besoin est celle des coûts. Un programme est toujours réalisé à l'intérieur d'une enveloppe financière donnée et il convient d'aboutir à une définition du matériel remplissant au mieux le besoin sans dépasser cette enveloppe.

En revanche la limitation des coûts peut être un moteur favorisant le progrès des applications technologiques. Elle oblige les ingénieurs à se remettre en cause et à constamment rechercher de nouvelles techniques plus adaptées pour assurer une fonction requise.

Tous ces éléments placent le responsable devant un dilemne difficile : celui de ne pas exprimer le besoin sous une forme qui conduirait à une réalisation excédent l'enveloppe financière, s'opposant à la crainte de manquer l'exploitation optimale des possibilités technologiques offertes.

Une autre difficulté réside dans le fait qu'il n'existe pas réellement dans les Etat-majors de personnels capables de combler le fossé qui existe entre l'opérationnel et l'ingénieur. Il faudrait en quelque sorte des opérationnels capables de se plonger dans la technique pour en éliminer tout ce qui est superflu ou inutilement coûteux et au contraire déceler toute application potentiellement efficace pour réaliser la mission.

1.5 - La charge de travail d'un équipage d'avion de combat

L'étude des conditions d'exécution des missions de l'aviation de combat moderne montre que ces missions vont en se compliquant dans un environnement opérationnel de plus en plus hostile et difficile à maîtriser.

De ce fait la charge de travail d'un équipage d'avion de combat va en augmentant de façon quasi explosive. Cet équipage doit tout à la fois :

- assurer le pilotage de l'avion,
- assurer le positionnement de l'avion et en particulier la navigation,
- maîtriser le mieux possible *la situation, ce qui signifie : percevoir*, identifier et synthétiser,
- décider de la tactique à court terme (la manoeuvre) et à long terme (la conduite de la mission),
- assurer l'auto-défense la plus efficace,
- gérer les moyens offensifs,
- gérer les senseurs en cherchant à la fois la meilleure efficacité et la discrétion,
- gérer les pannes et éventuellement les conséquences des coups reçus de l'adversaire.

Tout cela implique un excellent dialogue entre l'équipage et le système. Il faut donc étudier avec une particulière attention l'interface homme - machine : qualité des visualisations, viseur - visuel de casque, qualité des logiques système, commandes temps réel, commandes sensitives, informations et commandes vocales.

Par ailleurs la limitation des coûts de possession (c'est-à-dire à la fois des coûts d'acquisition et des coûts de fonctionnement) conduit à rechercher des avions de taille limitée servis par des personnels les moins nombreux possible. La tendance est donc d'aller vers des avions monoplaces où par conséquent le pilote doit exécuter seul l'ensemble des tâches présentées plus haut.

Pour que la charge de travail du pilote ne devienne pas rapidement impossible à gérer, il convient de faire assurer par le système une aide élaborée, d'où l'apparition du concept de "copilote électronique" et l'intérêt particulier porté aux techniques de l'intelligence artificielle et des systèmes experts.

2 - <u>LES TECHNIQUES DE L'INTELLIGENCE ARTIFICIELLE (IA) ET DES SYSTEMES EXPERTS (SE)</u>

2.1 - <u>De quoi s'agit-il ?</u>

La première difficulté que rencontre l'opérationnel en général et le responsable de l'expression du besoin en Etat-major en particulier, est de bien comprendre de quoi il s'agit quand sont abordés les problèmes liés aux techniques d'IA.

Il faut en effet constater que les spécialistes de l'IA n'ont pas toujours un langage clair pour faire comprendre les avantages de leurs techniques. Le terme même d'"intelligence artificielle", galvaudé par les journalistes, maintient à tord sur ces techniques un caractère ésotérique et il vaudrait mieux le remplacer par "<u>informatique cognitive</u>" (c'est-à-dire qui a trait à la connaissance).

Le <u>tableau 1</u> donne, par comparaison entre les techniques informatiques classiques et les techniques d'IA, les informations qu'il nous paraît nécessaire d'expliquer aux responsables pour leur faire saisir tout l'intérêt de l'approche IA.

2.2 - <u>Les domaines d'application</u>

Les techniques d'IA ont permis d'aborder des problèmes qu'il était jusqu'à présent impossible de traiter par l'informatique du fait de leur complexité. Les domaines d'application représentent donc un potentiel pratiquement infini, et il s'agit de sélectionner ceux qui présentent un intérêt pour les activités militaires.

2.2.1 - <u>Domaines en relation avec l'environnement</u>

Ce sont des domaines cherchant à reproduire des activités de base de l'intelligence humaine en rapport avec la perception. La réussite dans ces domaines est liée essentiellement à la puissance de calcul dont on peut disposer.

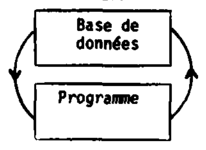
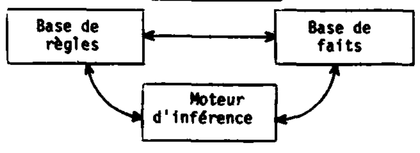
On y trouve essentiellement les deux domaines suivants :

- la reconnaissance de la parole, qui permettrait un dialogue vocal direct entre le pilote et le système,

- le traitement et l'interprétation d'images, la reconnaissance des formes et l'analyse des scènes.

<u>T A B L E A U 1</u>

TECHNIQUES INFORMATIQUES

APPROCHE CLASSIQUE	APPROCHE IA
- Programmation procédurale algorithmique c'est-à-dire où toute la connaissance est employée dans le même ordre en suivant pas-à-pas un enchaînement d'instructions appelé algorithme.	- Programmation déclarative qui consiste à déclarer toute ses connaissances sous forme de règles puis à ne les employer que quand elles sont nécessaires grâce à un raisonnement approprié.
- Instructions	- Règles, c'est-à-dire un formalisme traduisant les connaissances
- Algorithme assurant un ordre d'exécution des instructions	- Moteur d'inférence, c'est-à-dire un interprétateur de connaissances assurant le raisonnement
- Traitement de données	- Traitement de faits
- Procédé déterministe, c'est-à-dire utilisant toujours les mêmes déductions	- Procédé heuristique, c'est-à-dire cherchant à trouver la solution avec un raisonnement adapté aux faits à traiter
- Cherche la meilleure solution théorique	- Cherche la solution optimale en fonction des faits et du temps imparti
- Temps de traitement des problèmes complexes pouvant devenir explosif	- Temps de traitement des problèmes complexes pouvant être adapté aux délais impartis
<u>Système classique</u> 	<u>Système expert</u> 

2.2.2 - <u>Domaines des systèmes experts (SE)</u>

Ce sont les domaines présentant la potentialité la plus grande. Ils font pleinement appel à l'approche IA des techniques informatiques en utilisant les systèmes experts. La connaissance utilisée doit donc être recueillie auprès des experts militaires.

Signalons d'abord brièvement les domaines en rapport avec la conception et la fabrication des matériels aéronautiques car ils intéressent surtout les industriels. Cependant ces domaines doivent être suivis par les utilisateurs car ils ont des répercussions sur des activités dans lesquelles ces derniers sont partie prenante :

- l'architecture des systèmes,
- les études de sécurité,
- l'aide à la spécification technique.

Les domaines qui en revanche intéressent pleinement les utilisateurs sont les suivants :

- l'aide à la maintenance, le diagnostic de pannes,
- l'aide à l'interprétation des signaux radar et radio,
- l'aide à l'identification,
- la simulation, avec comme applications principales :
  - la simulation de l'activité de grandes unités militaires (divisions, unités aériennes, flottes maritimes),
  - la simulation de scénarios et de modes d'action stratégiques,
- <u>l'aide à la prise de décisions</u>, domaine extrêmement vaste dont les applications principales sont :
  - l'aide au commandement comprenant les activités suivantes : l'analyse stratégique et tactique, l'établissement de plans d'attaque et de défense, l'analyse et l'interprétation du renseignement, l'affectation des ressources,
  - l'aide à la conduite des systèmes dont les applications principales pour les avions de combat sont la préparation des missions et surtout toutes les activités embarquées couvertes par le vocable "copilote électronique".

2.2.3 - <u>Le copilote électronique</u>

Le but général des applications d'IA embarquée est tout d'abord d'alléger la charge de travail du pilote pour lui éviter un problème de saturation et ensuite de l'aider dans la réalisation de tâches complexes en particulier dans les prises de décisions. On devrait ainsi lui permettre de mieux se concentrer sur l'essentiel et donc d'être plus efficace.

De nombreuses activités, dites de "copilote électronique", peuvent faire l'objet d'applications IA plus ou moins complexes :

- la gestion de la mission et de la tactique à long terme,
- l'optimisation de la pénétration,
- l'aide aux manoeuvres d'engagement air-air,
- l'évaluation de l'environnement et la maîtrise de la situation,
- la gestion des capteurs,
- la surveillance des systèmes et leur reconfiguration,
- l'identification et la classification des menaces,
- la gestion des moyens de brouillage et de leurrage,
- la détermination de l'origine des pannes, de leurs conséquences, des actions curatives et les propositions de procédures d'urgence.

Toutes ces applications ont un point commun : elles doivent être réalisées sous forme de solutions <u>proposées</u> au pilote à l'exclusion de tout automatisme systématique, une proposition de solution erronée restant malheureusement toujours possible comme on le verra plus loin. Le pilote doit rester le seul juge de l'opportunité de la décision et les automatismes ne peuvent être envisagés que sur autorisation de celui-ci.

Une difficulté importante réside dans le manque d'aptitude actuel des systèmes experts à opérer en temps réel. Les décisions doivent en effet être prises très rapidement dans un avion de combat. Il y a donc des progrès à réaliser en matière de rapidité des calculateurs de bord et de leur adaptation aux procédés de traitement IA. Quoiqu'il en soit, il faudra toujours trouver un compromis entre le délai accordé à l'établissement d'une solution et la qualité de cette solution.

2.3 - Les avantages apportés par les systèmes experts (SE)

Il importe que le responsable de l'expression du besoin en Etat-major soit bien au fait des avantages apportés par les SE pour savoir quels sont les besoins qui peuvent être pris en compte par les techniques correspondantes.

- Les SE sont capables de traiter des problèmes qu'il était impossible de traiter auparavant du fait de leur complexité. Ceci répond bien aux problèmes de l'aviation de combat moderne : évolutivité croissante des situations, multiplication des informations, environnement hautement complexe et non structuré.

- Les SE sont capables de traiter des faits incertains.

- Les techniques des SE apportent à l'expertise : l'accumulation, la pérennité, l'ubiquité et la disponibilité.

- Les SE permettent d'assurer des dialogues homme - machine "intelligents".

- Les SE sont capables si nécessaire d'expliquer la raison des solutions qu'ils proposent.

- Les performances des SE ne sont affectées ni par le stress du combat, ni par la panique.

- Enfin les SE sont faciles à améliorer et à modifier.

2.4 - Les difficultés liées aux systèmes experts (SE)

Il faut en revanche être conscient d'un certain nombre de difficultés créées par les SE.

Tout d'abord un SE n'a de valeur que par la qualité de l'expertise qu'il contient. Cette expertise, forcément limitée par le nombre de règles qu'il est possible de mettre en oeuvre, peut être incomplète ou imparfaite. Les règles d'expertise peuvent être contestées selon les experts quand on aborde des sujets pour lesquels les solutions élémentaires ne sont pas évidentes, ce qui est souvent le cas en emploi opérationnel. Enfin les performances des SE se dégradent très rapidement quand on se trouve dans des conditions d'utilisation qui s'approchent des limites de l'expertise utilisée.

Pour toutes ces raisons un SE peut parfois donner une solution erronée ou tout au moins inopportune. Cela est-il acceptable sur un avion de combat ? Oui, à condition que les solutions proposées par les SE ne débouchent pas systématiquement sur des automatismes. C'est au pilote d'apprécier ces solutions et de les suivre ou d'autoriser des automatismes s'il l'estime nécessaire ou s'il n'a pas le temps de faire mieux.

Un SE est donc construit pour un expert capable d'apprécier la solution proposée et non pour un profane.

D'autres difficultés résident dans le recueil de l'expertise :

- ce recueil est assuré en faisant interroger des experts par des spécialistes (en général psychologues) ce qui est un travail long et fastidieux demandant une totale disponibilité des experts,

- la formalisation des connaissances sous forme de règles limitées en nombre n'est pas une tâche aisée,

- la transmission de la connaissance par les experts se heurte parfois à des obstacles psychologiques de refus,

- enfin se posent les problèmes de propriété, de protection et de non-divulgation de l'expertise.

Pour ces raisons il faut accorder de l'importance au développement des techniques de génie cognitique (spécialisé dans le recueil des connaissances).

2.5 - Les techniques à développer

Pour répondre aux besoins des utilisateurs en matière d'IA, il convient donc de connaître les techniques qui doivent encore progresser :

- amélioration des logiciels, traitement symbolique des faits, langages adaptés à l'IA,
- raisonnement sur l'incertain, logiques floues, concepts de possible et de nécessaire,
- rapidité d'exécution à améliorer pour atteindre le temps réel,

- calculateurs spécifiques IA, embarquabilité,
- capacité des mémoires de masse,
- traitement de la priorité,
- coopération entre plusieurs SE,
- formalisation de la connaissance, génie cognitique, apprentissage automatique.

3 - LES SOLUTIONS

Le travail du responsable de l'expression du besoin en Etat-major se heurte couramment, comme nous l'avons vu, à de nombreuses difficultés pour prévoir les conditions dans lesquelles s'exécuteront les missions dans l'avenir.

Quand ces conditions font en plus appel aux techniques de l'IA, ces difficultés sont aggravées par des incertitudes dues à une information insuffisante, donc à une méconnaissance des problèmes et de leurs solutions potentielles, et à une organisation inadaptée.

Les domaines qui suivent doivent donc faire l'objet d'un effort particulier de la part des Etat-majors.

3.1 - Choix des applications IA

Pour être adaptée au traitement par les techniques IA, une application doit répondre à plusieurs critères.

Tout d'abord le problème à traiter :

- doit être suffisamment complexe,
- ne doit pas avoir de solution algorithmique, ou cette solution doit demander un temps de calcul excessif,
- ne doit pas avoir de solution intuitive,
- est particulièrement indiqué si les informations sont incomplètes ou imprécises,
- est également indiqué si les règles de la base de connaissances évoluent rapidement.

Ensuite le recueil de l'expertise :

- doit être effectué auprès de praticiens expérimentés,
- et ces praticiens doivent être totalement disponibles.

3.2 - Maîtriser les SE

La maîtrise des SE ne sera effective que si les responsables en Etat-major sont conscients d'un certain nombre de problèmes.

- Comme nous l'avons vu précédemment, un SE n'a de valeur que s'il est réalisé pour être utilisé par un expert pouvant le contrôler et non par un profane.

- Il faut savoir mesurer ses ambitions : un SE disposant d'une centaine de règles est un optimum aujourd'hui. En particulier la réalisation d'un "copilote électronique" ne pourra se faire qu'à travers une approche prudente par "petits pas".

- Il faut savoir trouver les concepteurs les plus compétents pour le SE envisagé. La compétence dans les techniques IA est en effet souvent très diversifiée : universités, organismes de recherche étatiques, industriels.

- La réalisation d'un SE représente en général un coût limité au développement. En revanche les délais de mise au point peuvent être très importants.

- Les simulations pilotées, déjà essentielles pour le développement des systèmes d'armes, deviennent indispensables pour la mise au point des systèmes experts embarqués.

- Ce sont les utilisateurs, et non les industriels, qui doivent être responsables de la maîtrise de la partie expertise. Pour cela il faut disposer de personnels totalement disponibles.

- Les utilisateurs doivent également faire évoluer eux-mêmes la base des connaissances.

- Il faut donc que les utilisateurs disposent d'une organisation adaptée.

3.3 - Une organisation adaptée

Cette organisation doit être capable de résoudre les problèmes de compétence et de recueil de l'expertise.

La compétence doit être assurée par la formation IA des personnels suivants :

- au niveau de l'Etat-major, responsable des programmes, une équipe IA composée d'officiers de différentes spécialités,
- dans les organismes d'essai, d'évaluation et d'expérimentation, au moins un officier compétent dans chaque organisme.

Comme nous l'avons vu, le recueil de l'expertise constitue une charge de travail importante et requiert une totale disponibilité. Ce recueil doit être :

- ordonné et dirigé par l'équipe IA de l'Etat-major, directrice des programmes,
- réalisé par des experts choisis et disponibles sous la responsabilité des officiers compétents dans les différents organismes.

Par ailleurs, les problèmes de protection de l'expertise doivent être traités par l'équipe IA de l'Etat-major.

4 - CONCLUSION

Les techniques de l'intelligence artificielle portent en elles un potentiel considérable d'efficacité pour les matériels aéronautiques militaires du futur.

Le responsable de l'expression du besoin en Etat-major doit en tenir compte, ce qui implique des actions en matière d'acquisition de la compétence et d'organisation pour que les choix correspondants et le suivi des réalisations soient effectués correctement.

SOFTWARE PRODUCTIVITY THROUGH Ada* ENGINES

Capt Rick A. Long, USAF
AFWAL/AAAF-2
Wright-Patterson AFB, OH 45433-6543
(513)255-2446 AV 785-2446

This paper explains how the use of correct computer hardware can be used to increase programmer productivity. It uses an Air Force program for specific information and examples.

TRADITIONAL PRODUCTIVITY VIEWS

When one thinks about software productivity, one tends to think about the more traditional methods of increasing productivity; high-powered software tools coupled with high-powered computers. Syntax directed editors for most languages have been around for a long time now. Program generators have become almost commonplace. Even a popular $39.95 Pascal compiler that has several screen-entry program generators available.

Recently, there has been a new wave of software productivity tools. They all use very sophisticated software coupled with a reasonably high-powered computer. A good example is the Interactive Ada Workstation (IAW) which General Electric is developing for the Air Force. This software development tool constructs a compilable Ada program with the aid of graphical representations (e.g. finite state machines and Buhr diagrams). The IAW runs on a $100,000 per user lisp processor. The productivity gain of the IAW is estimated to be at least one order of magnitude beyond ordinary software engineering techniques. This is an effective tool to assist in software productivity increases; but, it still fits the traditional high-powered computer running a high-powered program.

What happens when a programmer uses one of these traditional software productivity tools to program an embedded computer? This is where the problems start with respect to traditional software productivity views. From the outside, it looks as though there are no problems at all. In fact there can be enough problems created to rip apart any previous notions of programming. Let's take a look at how this develops.

First there is the super software development tools. Ada has several features which are conducive to software productivity gains. One can reuse previously written code, even if it varies slightly from what they really need, through the use of generics. Many people argue that reusing code in the avionics environment is not practical. However, for many other applications it makes sense and will save development costs. A generic package for doing screen manipulations for terminals (which is used in almost all Ada programs) could result in cost savings if implemented and used. It is doubtful that anyone would (or could) argue that a database package is not a good candidate for reusability. Many applications are good candidates for code reusability.

Another powerful feature of Ada is tasking. True tasking can be very powerful. Ada has given the programmer the flexibility to write programs that run simultaneously without having to think about how tasks are scheduled or how they communicate, as is required in other languages; that is built into Ada itself. All of the built in capabilities of Ada tasking also causes code size and execution speed problems. Previously, if it was not written in assembly language, the tasking feature required a specially written run-time system to handle its intricacies and allow reasonable execution speed. The problem here seems to be due to the compiler implementations.

The next feature of Ada is exception handling. Formerly, a programmer had to write exception handling features separately, frequently in assembly language. Ada does not require that the programmer write in this capability; the language does it for them. The problem with exception handling, however, is that it adds overhead to nearly all of the other features that Ada provides. During every procedure call, for instance, Ada checks for any imaginable error or inconsistency. Programmers have the option of turning off error checking, but they may not want to.

The final feature (or non-feature) is precise timing. Ada just does not allow for it. The programmer is not allowed to force an event to occur at a specific time. The only timing feature is the delay statement. ANSI/MIL-STD-1815A defines the delay statement in the following manner: "and suspends further execution of the task that executes the delay statement, for at least the duration specified by the resulting value." In other words, "delay at least this long." How does one implement the cyclic executive without precise delays? Probably in assembler or a more difficult high-level language like FORTRAN.

* - Ada Is A Registered Trademark Of The U.S. Government (Ada Joint Program Office) 1.

The next super software tool is the type that resembles the Interactive Ada Workstation (IAW) mentioned earlier. It doesn't require one to know a particular programming language (in this case it is Ada) in order to develop the required software. All you need to know is the basic design of the program. The IAW is a super software product, but if it were not running on a super computer (1 million instructions per second or MIP) per user it would not be nearly as useful as it currently is.

Secondly there is the super computer. The definition of a super computer, in this context, is one that is capable of giving the programmer a reasonable response time when running reasonably sophisticated software. It has been found that programmer productivity is directly related to computer response time. The response time should be, obviously, as short as possible. It should, also, be consistent. Imagine working on a program in which the programmer just instructed the computer to compile a program. The first time it was compiled the prompt was returned in two minutes. The programmer then finds that there is an error in the logic so the programmer must edit and recompile the program. This time the computer returns in twenty minutes. But, it was expected to return in two. The programmer must sit and wait for it to return with no idea about how long it will take. In both cases the time was wasted. These are just two reasons that most software productivity systems are hosted on a super computer. Examples are the Lisp Machines, Inc. (LMI) and Symbolics lisp processors, the IBM 4341, the DEC VAX, the Sun workstation and even the IBM AT.

PROGRAMMING PRODUCTIVITY AND EMBEDDED (SMALL) SYSTEMS

Everyone concerned with Ada's use in embedded systems knows the problems with using Ada in that arena. Actually one can't blame Ada; the problem lies with the particular implementations of the compiler. The problem is compounded when using the MIL-STD-1750A computer. It is slow when compared with the super computers mentioned above. The 1750A computer has a very limited memory capacity. Used in embedded systems, the 1750A computer tends to be used in critical real-time systems such as the Advanced Tactical Fighter (ATF). Pilots lives depend on the effectiveness of the computers on the airplane. Knowing how much memory an Ada program with tasking and all its associated error checking and correction requires, as well as the processing power required, one tends to not use the features of Ada mentioned above even though the promise is great.

So what does one do if one wants (or is required) to use Ada to program embedded systems. Well, mostly, only use the parts of Ada that the 1750A is capable of handling. But then the programmer might as well have used FORTRAN; hence the name AdaTRAN. When programmers can't get around using particular Ada capabilities they are forced to take other steps.

There is a lot of overhead associated with Ada tasking. The run-time required to support this feature can fill a lot of memory that the 1750A just does not have. However, the more important problem with tasking is speed. Task switching takes a lot of time to accomplish. Embedded systems are not the place to have a slow task switch. Task switching in cyclic executives frequently happens 60 times per second. In some compiler implementations the whole duty cycle would be taken up with task switching, not doing any processing. Two of the things that a programmer can do to remedy this problem are just not use Ada tasking or, if they have to use Ada, (usually the case) just rewrite the run-time to be fast enough to be usable. Down goes the productivity rate.

The cyclic executive was mentioned above. Ada does not allow for the cyclic executive which requires an accurate clock. Ada does not have a means for accurate timing. One must modify the run-time again, or even worse. One might have to go back to school to learn different schemes to control the system that they were working on. Again, down goes the productivity.

Going hand-in-hand with the cyclic executive problem is the control theory problem. Traditional control algorithms require that a program implementation have access to precise timing; Ada does not have it. Again, it is back to school or the drawing board to modify the run-time and again the productivity suffers.

Next comes the idea of using Ada tasking for distributed processing. If one values their job, don't mention this idea. To date there are no compilers with the capability to produce code for the 1750A that allows it to be in a distributed environment. When there is one available it will be for a specific architecture; probably not the one needed. If distributed processing using Ada is an absolute must, then a lot of money and time writing a compiler (or having someone else write it for you) will be spent. One more time, down goes the productivity.

Hopefully by now I am sounding a little like a broken record. You are also probably putting two and two together and coming up with the idea that it not only takes super software coupled with a super computer to maximize software productivity, but it also takes a target computer capable of handling the language features that is required for the implementation.

A SOLUTION

If the problem is that the limitations of the target computer results in the programmer becoming less productive when writing programs what should, be done about it. It may well be argued that the standards must be changed. Maybe the Ada standard should be changed. There are those who advocate that the embedded computer standard should be changed; use a more powerful 1750 computer. Still others say that the military should use a completely different computer standard. Still others say that Ada should be used as the overriding standard and allow the implementor to choose the computer system that can handle the implementation. The answers seem to be pointing in the same direction. They seem to be saying that the super productivity tools available should be retained; it is the hardware that must be change.

One solution is to simply to change the 1750A; to beef it up. This might be done by taking the 1750A and adding the required hardware (coprocessor or accelerator) to give it the required capability to allow it to compete with the super computer. Give it the capability to do the things that the executing Ada program normally does very slowly in software, in hardware. Then the programmer is allowed to use the pieces of Ada that can increase their productivity without thinking about the hardware implementation.

AN EXAMPLE

Recently, the Air Force Wright Aeronautical Laboratories (AFWAL) had a contract through the Small Business Innovative Research (SBIR) office to do an Ada primitives study. The goal was to find out how to increase the speed of an executing Ada program by looking at the operating system primitives most used. Among the methods of increasing the speed recommended was the Ada coprocessor, or Ada accelerator. It would serve much like the math coprocessor chip does on the normal microcomputer but it would be for the specific purpose of making Ada run faster.

Four main Ada primitives were recommended to be put on the coprocessor chip/board. 1) Tasking management and switching. This area holds the most promise for speed increase. 2) Timer services. This would add a capability that does not now exist. 3) List manipulation. This goes along with task management. 4) Heap storage management. These four areas will be discussed in greater depth later.

The main suggestion of the SBIR was to build a coprocessor board or chip. The coprocessor would accompany a Motorola 68000, Intel 8086/80286 or 1750A based system. The coprocessor should be designed so that it can be transferred from one system to another as easily as possible. Once the hardware has been prototyped, an existing Ada compiler should be modified to produce the required code for the target system.

PAYOFF (HARDWARE)

The SBIR contractor implemented one of the suggested Ada primitives (list manipulation) and simulated it on a silicon compiler. Work with the silicon compiler demonstrates that one could expect a 16 MHz coprocessor clock rate and it suggests that up to 24 MHz may be possible. The simulation was of a simple linked list fetch. A linked list software example was also designed and implemented in C on an Intel 8088 development system hosted on a VAX 11/750. The development system emulator was then used to determine the number of clock cycles required for each instruction. This data was used to estimated the overall timing. The results were quite staggering.

The results showed that much can be gained using a coprocessor. Using a 5 MHz central processing unit (CPU) (like the IBM PC) with a 5 MHz coprocessor, the time it took to find the nth block whose ith word contains a specific value was estimated. Using 100 and 400 for n, it was found that the silicon version worked an average of 167 times as fast as the software version. If a 16 MHz coprocessor is used with the same 5 MHz CPU one can expect a speed increase of over 500 times. Since task switching is much more complex, the payoff of implementing it in silicon could potentially result in a higher payoff. Adding the four primitives mentioned, the payoff could be staggering in terms of both hardware performance and software productivity.

PAYOFF (SOFTWARE)

Increase in software productivity resulting from the increase in hardware performance inevitable. With the increases of performance in existing Ada capabilities coupled with the addition of precise timing one claims many advantages over traditional methods of Ada programming.

The largest advantage could be the elimination of the need to rewrite the executive. Since tasking works fast enough now and has available garbage collection, one no longer needs to modify the existing executive to gain these advantages. To gain speed increases in tasking, the programmer might try to switch off some of the error checking that Ada automatically uses -- losing some of the programmer's confidence in the program. That will no longer need to happen.

No longer will the programmer be forced to program in assembly language (at least ideally not) no performance critical sections of code. Studies show that the average number of lines of code per programmer per day over the life of a software development project is constant; regardless of the programming language being used. If programmers can use a higher percentage of Ada and a smaller percentage of assembler, the amount of effective work that each line of software does has increased. An automatic increase in productivity is realized.

Another benefit is that the programmer does not have to learn new methods of control theory. Generally, control theory requires that one knows exact times between occurrences of events. Ada does not support this capability. Programmers have to use other languages or different control theories; theories that are not familiar to them. It requires them to learn the different theories -- taking time from the project. The inclusion of hardware supported precise timing allows the programmer to use familiar ideas and, again, will result in increased productivity.

Allowing a programmer to use Ada with no other language mixed in results in another great benefit. It makes use of the methodologies that the designers of Ada had in mind. Ada was created with many software design methodologies in mind -- functional decomposition, top-down and object oriented design being three. Taking away any of the capabilities of Ada requires the programmer to write "kludges" and "patches" detracting from the readability, supportability and maintainability of the program. While these three advantages do not fit strictly in the productivity mold, they do fit nicely into another mold -- the life-cycle cost mold. Not only will the original programmer have a higher productivity rate, the person maintaining the program will also have a higher productivity rate. Maintenance costs are particularly important since maintenance results in as much as 95% of the total life-cycle cost of the software.

CONCLUSION

To reiterate, software productivity tools have evolved to a very high state. The tools are generally associated with one of the super computer defined earlier in the paper. The problem arises when the programmer can't use all of the facilities of the language because of the limitations of the target computer (and in some cases the limitations of the language).

The conclusion is very simple: In order to achieve and maintain a high software productivity rate, adequate target computers are needed as well as adequate software development tools.

ACRONYMS

Ada..............	Not an acronym
AFWAL............	Air Force Wright Aeronautical Laboratories
ATF..............	Advanced Tactical Fighter
CPU..............	Central Processing Unit
IAW..............	Interactive Ada Workstation
MIP..............	Million Instructions Per Second
SBIR.............	Small Business Innovative Research

SOFTWARE DEVELOPMENT GUIDELINES

By

Denice S. Jacobs
Aeronautical Systems Division
Air Force Wright Aeronautical Laboratories
Wright-Patterson AFB OH 45433-6543
USA

SUMMARY

Due to the growing complexity of avionic systems, the development cycle for mission critical software has evolved into a collective process of organized tasks. These tasks are distinct levels of effort which are implemented by the developer to ensure the creation of a reliable, operational system. This paper summarizes four principle tasks which have proven to be excellent procedures for developing avionics software. The first and foremost task of the project manager is to establish a Configuration Control Board (CCB) as the central core of technical management. It consists of a group of key hardware and software engineers who mutually govern the status of system development, and incorporate design changes on an agreed-to basis. The second task is to logically separate the software project into well-defined phases of development. This, too, requires the cooperation of both hardware and software teams to work together in accordance with a master schedule. The third task is to create an automated data base which contains the latest interface specifications (ICDs) and system message definitions for use by the engineers. Finally, the last task is to procure hardware emulators and stand-alone test stations as an effective means of testing software prior to system integration and test (I&T).

INTRODUCTION

Due to the growing complexity of avionic systems, the development cycle for mission critical software has become more involved in terms of the difficulty and number of software, interface, and test requirements to be defined. Consequently, the probability of incurring design/coding errors is increased due to written and verbal miscommunications between the hardware and software development groups, information latencies due to the number of developers involved, and inadequate software testing prior to system I&T. In order to help minimize the aforementioned problems, it was necessary to create several guidelines which 1) enforce a formal communication network between the hardware and software groups; 2) enhance the visibility of software progress; 3) improve information access; and 4) enhance the software verification process. These guidelines evolved into four specific tasks which are addressed in the following paragraphs.

TASK 1: CCB MANAGEMENT

The first task is to establish a Configuration Control Board (CCB) which enforces a formal communication network between the hardware and software development groups. The Board consists of key engineers and managers from both disciplines who mutually govern the status of the system and incorporate design changes on an agreed-to basis. They attend all formal design reviews and major design walk-throughs to ensure that neither group is being compromised. They also meet on a weekly basis to review the current state of the system and examine design problem reports which are generated throughout the development effort (i.e., from requirements definition to system I&T). Finally, all design-related issues are properly documented and reviewed by the CCB before any changes are officially made to either the hardware or software designs.

TASK 2: SOFTWARE DEVELOPMENT PHASES

The second task is to divide the software project into six logical phases of development:

1) Software Requirements Definition
2) Top-Level and Detailed Design
3) Module Code and Test
4) Component I&T
5) Configuration Item I&T
6) System I&T

The first phase is to define the software requirements in relationship to the hardware requirements to ensure that design oversites are not incurred. Additionally, this phase must be completed prior to starting the second phase in order to establish a known baseline of requirements.

The second phase mandates the satisfactory completion of all of the software design reviews and walk-throughs prior to coding the software. Work folders are then established which contain all of the software material pertaining to a particular processing element called a configuration item (e. g., data processors and signal processors are two separate configuration items). Examples of data recorded in the work folder include software requirements, top-level and detailed design descriptions, walk-through results, source listings, module code and test history, component and configuration item I&T history, and design problem reports.

The third phase requires that each software function be developed in a modular fashion and tested accordingly (i.e., a module is defined as being a small compilable file which performs a single function in an efficient manner). In addition, module development is typically performed on a mainframe computer so that the programmer can take advantage of various support tools such as the Scenario Generator/Monitor, the ICD data base and debuggers.

Once each module meets its own unique performance requirements, then software development transitions to the component I&T phase where a set of related modules are integrated and tested as a whole entity. Again, this phase of development takes place on a mainframe to take advantage of the support tools.

The next phase is to integrate and test a set of related components which comprise a given configuration item. This development takes place on both the mainframe and the target processor so that other helpful tools may be utilized (e.g., the hardware emulator which resides on the mainframe and the stand-alone test station which interfaces to the processor). Additionally, the developer will be able to assess how well the software runs on the hardware during this phase of development.

The final phase is System I&T, whereby the configuration items are integrated together to create individual threads of operation. This is the final and most difficult stage of software development since it involves the integration of several unique hardware and software elements.

In summary, these software development phases have proven to be excellent procedures for developing software while, at the same time, enhancing program visibility and control.

TASK 3: AUTOMATED ICD DATA BASE

The use of an automated ICD data base is beneficial to the user in several respects. By its very nature, it is readily available to anyone who has the appropriate need-to-know authorization to access the host computer which supports the toolset. The data base also contains a detailed description of every message and ICD used within the system. Therefore, written and verbal miscommunications between the various development groups are effectively minimized, if not totally eliminated.

The data base is maintained by the CCB on a weekly basis. Any proposed changes to the baselined hardware or software designs are always documented in design problem reports and formally reviewed by the CCB. If approval is granted, then the data base is updated by the CCB to reflect these modifications.

In conclusion, the automated data base is an invaluable source of design/interface data because it is readily accessible to all users and provides current information.

TASK 4: SOFTWARE TEST ENVIRONMENT

The following test environments were chosen since they verify three aspects of software design, namely the software-to-software interfaces, the hardware-to-software interfaces, and real-time operation.

Scenario generators and monitors are useful software programs which are used to stimulate the software packages while monitoring the message traffic between the software modules (i.e., software-to-software interface). Hence, the developer can easily test the software with a number of possible scenarios and examine the resultant messages.

Hardware emulators are also useful software programs which emulate the operational characteristics of the host processor. This is beneficial in determining the validity of the software algorithm in relation to the processor's capabilities and limitations (i.e., hardware-to-software interface).

Finally, the stand-alone test stations are unique hardware environments which are used to debug the software during real-time operation. This, too, is an invaluable tool since it permits the developer to examine the results from individual software instructions during actual execution times.

CONCLUSION

With the advent of highly programmable systems, there surfaced a need within the avionics community to establish guidelines which definitized the relationship between hardware and software development efforts. These guidelines evolved into four specific tasks which ensure that: 1) design oversites are minimized; 2) software progress is controlled; 3) critical interface definitions are readily available; and 4) software is fully tested prior to system I&T. It is believed, therefore, that the combined implementation of all four tasks will ensure the development of a reliable, operational system.

DISCUSSION

K.Benner, US

Is the government involved in the weekly configuration board meetings?

Author's Reply

The weekly Configuration Control Board (CCB) meeting was in reference to the contractor's software development. Many times a particular effort will involve a prime contractor and several subcontractors. The contractor's CCB facilitates the development of software when more than one company is involved.

ON THE CONDITIONS AND LIMITS OF USER INTERVENTION IN DELIVERED SOFTWARE MANUFACTURER'S VIEWPOINT

Jean R. CHARVOZ
Software Engineering Manager
THOMSON-CSF RCM DIVISION
178, Bd Gabriel Péri
92240 MALAKOFF

ABSTRACT

High capacity data processing architectures are now available in avionics. The development of embedded software has become a major activity in most avionics Equipment manufacturing Companies.

These companies are faced with challenges. They have to master the quickly evolving software technology and to develop the associated know-how. They must also face a change in the internal economic structure of the company and in the financial relationship with the customer resulting from the increased incorporation of software in the equipment.

Some features inherent in the software reinforce the idea that changes in the functions performed by an equipement will be implemented more easily. Many customers ask for facilities allowing them to modify software themselves. These requests often result from the customer's desire to acquire a degree of independence with respect to the manufacturer.

To meet these requirements, the manufacturer's position has to satisfy different, and sometimes contradictory, conditions :

. the desire to keep meeting customer's needs,

. the need to maintain and improve a technologically leading position and to ensure adequate financial resources,

. the need to maintain and improve its own image.

The first part of the paper is a tentative answer to the following questions :

. what characterizes the development of an equipment which includes software : types of software, activities, tasks, skills, facilities.

. why are software modifications equivalent to a partial redevelopment and where do the difficulties lie,

. what are the conditions under which a customer may modifiy the software of an equipment.

The second part of the paper provides the answer to some aspects of the process of giving a user the autonomy to perform software modifications.

PART 1

1 - SOME CHARACTERISTICS THAT INFLUENCE SOFTWARE IN AVIONICS

Software incorporated in on-board military equipment or systems generally has to work within tight constraints. According to the definition in (BOEHM 81) this software must operate in a strongly coupled complex of hardware, software and operational procedures.

The requirements of the equipements and systems, expressed in terms of maximum volume, weight, power dissipation, performance, and functional complexity, result in severe constraints.

The final definition of the embedded software results from optimization of the design of the equipment or system in order to satisfy as far as possible the preceding constraints.

The long operational life of equipment and systems is another factor which greatly impacts the definition of software : the cost of changing the hardware is so high that the software is expected wherever possible to handle the changes whatever their origin (problem solving or functional updating).

2 - PARADIGM FOR THE DEVELOPMENT OF THE SOFTWARE OF AN EQUIPMENT

The waterfall model of software life-cycle has been widely described ([ROYCE 70], [BOEHM 81]) and is still a reference. A useful variant of the model considers that the integration and test phase should be divided into sub-phases. Each of these sub-phases is assigned a precise objective of verification and validation of the results of a corresponding analysis or design activity. This variant model is in the shape of a capital V.
All these models apply to the development of embedded software except for one characteristic :

the software is not the final product, but only part of the whole equipment or system.

The avionics equipment or system manufacturer needs a broader model applicable in the development of systems. This model may be obtained by extending the previously introduced V form software life-cycle.

A useful example of sucl. a model has been introduced by [ITI 85] (see figure 1).

The activities involved in the development of an equipment may be concisely defined as follows :

. **Equipment Requirements analysis**

- consists of determination, specification and review of equipment functional, performance, interface and verification requirements,

- Results in :

  . approved, validated equipment requirement specifications validated for completeness, consistency, testability, feasibility,

  . approved, validated concept of operation including preliminary user's manual,

  . development plan : milestones, resources, organization, responsibilities, activities, techniques, products,

  . control plan : configuration management plan, quality Assurance plan, overall test plan,

  . definition of overall and in-flight tests.

. **Equipment Design phase**

  - Consists of : determination, specification and review of hardware-software architecture,

  - Results in :

    . validated equipment design specification : hardware, software, hardware - software interface specification, verified for completeness, consistency, feasibility and traceability to requirements,

    . identification of high-risk development issues,

    . preliminary integration and test plans.

    . definition of needed specific testing equipments.

- **NOTES** :

  The software definition clearly appears as the result of an equipment design activity.

  Due to the optimization process introduced in paragraph 1, the equipment or system design activity is carried out with iterations. In many cases, a partial prototype must be built for feasibility study purposes.

  The design decisions made during this phase are very important and must be carefully documented in order to ensure their traceability.

3 - <u>TYPES OF SOFTWARE INCLUDED IN AN EQUIPMENT</u>

The software included in an equipment or a system typically perform functions related to :

A - signal processing,

B - control of the equipment or of the system,

C - information processing.
In the A and B fields and to a lesser extent in the C field, the definition of the software is strongly affected by the constraints and by the optimization process which have been introduced previously.

In some parts of the B field and in the C field, the definition of the software may be more easily related to the operational purpose of the equipment or of the system.

Two types of software can be distinguish ed :

. Embedded firmware is characterized among others things by the following features :

- the functions performed suit the hardware and are often related to physical effects : the choice of data organization and algorithms is based on thorough scientific and experimental knowledge and know-how,

- the implementation often requires the use of microprogramming, assembly language, optimization. The result is a combination of a hardware device and software that lies as read only software on the hardware device.

. operational software

- the functions performed suit the user's operational requirements. Operational information rather than primary data is manipulated. The choice of data organization and algorithms also requires thorough knowledge and experience, but these are easier to transfer,

- the implementation is based on the use of a higher order language. It is less hardware dependent and more subject to changes.

4 - ABOUT TESTING

In most cases, the criticity of the functions performed by the equipment or sytem result at software level in severe testing requirements.
The main difference with the development of non-embedded software lies in the need for particular instrumentation and/or of specific test equipments.

In general, these test equipments are software driven. This software depends on :

. the architecture of the test equipment,

. the functions to be tested (black box testing),

. the organization of the equipment or system to be tested (white box testing).

It follows that any modification of the functional definition and/or the design of the equipment will have a significant impact on the definition of the software of the test equipments.

Testing is an activity essentially oriented towards detection of failures and defects, the goal being to ensure correctness and reliability [TURING 50] [BRANDSTAD 80].

Once an error is detected,it must be fixed. Real time aspects make this activity difficult and most often some special instrumentation is required.

This overall activity must be carefully managed under two aspects :

. configuration management,

. identification and updating of a convenient subset of test data (the non-regression subset), the goal being to check for undesirable effects of modifications without re-executing all tests previously executed.

5 - ABOUT SOFTWARE DEVELOPMENT FACILITIES

These facilities consist of computers and software tools.

Their role is to help the developer in the various aspects of his activity :

. technical aspects : e.g. : design tools, compilers, generators,

. documentation tools,

. management tools,

. librarians, long term storage...

These tools are mainly software tools. They may be updated and this results in specific software engineering activity :

. configuration management of tools,

. determination of the tests to be performed in order to decide whether or not a new release of a software engineering tool may be used to modify a product of an earlier release.

6 - ABOUT KNOWLEDGE, SKILLS AND KNOW-HOW

The equipment manufacturer must carry out development activities for his own use and increase his knowledge and know-how in a lot of areas. Those which are of interest here are :

A - Operational area

e.g. in the Electronic Warfare field :

. general and theoretical knowledge concerning detection analysis, jamming techniques and coordination as well as threats.

. definition of operational and technical requirements,

. pre-flight preparation, analysis of flight report readouts.

B - <u>Equipment or System Design and Testing</u>

. design techniques, data processing architectures, signal processing algorithms, design verification and validation techniques (e.g. simulation, prototyping...),

. equipment and system integration and testing techniques, testing equipments design,

. in-flight testing techniques.

C - <u>Software Engineering</u>

. methods and techniques used to support the software development activites :

Software requirements analysis

Design : determination of software architecture, program design and data base design.

Programming...

. documentation : development and update of :

- user documentation,

- development and maintenance documentation.

D - <u>Facilities management</u>

. operation of development facilities,

. updating of tools over a long period of time,

. configuration management of tools.

6 - RELATIONSHIP BETWEEN TYPES OF SOFTWARE AND KNOW-HOW

TYPE / KNOW-HOW	FIRMWARE	OPERATIONAL SOFTWARE
OPERATIONAL AREA	+	+ +
EQUIPMENT DESIGN & TESTING	+ + +	+
SOFTWARE ENGINEERING	.	.

.   Normal skill required

+   Extensive coupling : high skill required

++   Very extensive coupling = very high skill required

+++ Extremely extensive coupling = extremely high skill required

7 - ABOUT THE DOCUMENTATION OF SOFTWARE

The very many guides and standards that exist implie the necessity for a large quantity of documents to be produced throughout the whole development phase.

All this documentation is very useful to support the quality assurance activities as well as the development activities.

This results in a lot of cabinets full of documentation and the question is : How really useful is all this paper (or magnetic media) to support updating of the software in the long term perspective

The main problem concerning the documentation lies in the lack of :

. completeness,

. readability : it is a matter of precision, conciseness.

The many modifications made during the development phase especially during integration and testing, affect readability as well as completeness.

. usefulnesss : the documentation should not only describe the software architecture, but also why this architecture has been chosen and how it has been tested. Facilities should be provided to facilitate access to a given information.

Most often the required information exists, but is distributed across hundreds of pages of documentation resulting in low usefulness. It is a matter of documentation and presentation.

Few people are able -or want- to study such documentation, and even fewer customers are willing to pay the real price for it !

The documentation should be elaborated in two steps :

. during the development phase,

. at the end of the development phase, a maintenance documentation should be elaborated and verified.

As this documentation is bound to be updated, this process must be aided by strong methodologies and tools.

8 - CONCLUSIONS OF PART 1

CONCLUSION 1 :

The conclusion is that updating of firmware is much more difficult and risky to bring out than updating of operational software is.

Thus the equipment manufacturer's position is that only modifications to operational software are to be proposed.

CONCLUSION 2 :

The preparation and updating of the maintenance documentation is a costly process.

The resulting documentation includes a large amount of the manufacturer's know-how. It leads to a large saving in maintenance costs. Supplying this documentation is equivalent to a know-how transfer. Thus the manufacturer's position is that there should be a negotiation on this basis at contract time.

CONCLUSION 3 :

Part 1 has shown that any intervention on the software requires the existence in the user's country of the same facilities as those of the manufacturer's and the existence of the appropriate set of skills.

Even if the manufacturer cannot accept to guarantee the results of the user's intervention, his desire is to keep meeting customer's needs.

The only solution is to aid the user to acquire autonomy through a cooperation process. Because this process implies some form of technology transfer, it will be designated as the transfer process.

Part 2 provides a broad definition of this transfer process as it applies at the present time between THOMSON-CSF and some of its customers.

PART 2

- TRANSFER PROCESS -

The purpose is to develop autonomous software updating capabilities in the customer's country.

The process involves four phases :

Phase 1 : Transfer requirement analysis and definition of the cooperation phase.

Phase 2 : Initialisation.

Phase 3 : Cooperation.

Phase 4 : Autonomy.

Phase 1 - End points

. approved, validated concepts of operation of the original equipment or system,

. approved, validated identification of the operational software,

. definition of the scope of concept updating permitted by modification of the operational software,

. approved choice of the updating to be undertaken in the cooperation phase,

. approved identification of the currently existing human resources and know-how in the customer's country,

. approved definition of the necessary additional resources (human, know-how, computers, instrumentation, software, engineering tools...),

. approved definition of the training,

. approved definition of the additional supplies (development, documentation, partial and overall specific testing equipments etc.).

. approved definition of the top level co-development plan, including milestones, resources, responsibilities, schedules and major activities, overall work-sharing between customer and manufacturer,

. approved co-development contract based on the above items.

Phase 2 - End points

. delivered and installed operational equipment or system,

. completed training of the different categories of personnel,

. completed installation of the additional resources identified in phase 1 (software development center, specific testing equipments, flight testing base...),

. approved, validated definition of the equipment or system updating to be developed in the co-development phase,

. approved, validated software requirement specification : functional, performance specifications validated for consistency, testability and feasibility considering the existing hardware - software interface as a constraint,

. detailed development plan :

- milestones, resources,

- organization and responsibilities. Precise details of work-sharing between Customer's and manufacturer's teams,

- activities, schedules, products,

. detailed control plan : configuration management, quality assurance.

<u>Phase 3</u> - **<u>End points</u>** : Completion of System Acceptance Review

. Concerning the software :

- Satisfaction of software Acceptance tests.

  Verification of satisfaction of software requirements.

- Acceptance of the modified deliverable software products : documentation.

. Concerning the system :

Satisfaction of System Acceptance test :

- verification of satisfaction of systems requirements,

- verification of operational readiness of system, facilities and personnel.

Acceptance of the modified deliverable system products : hardware, software, documentation, training, facilities.

Completion of all specified conversion and installation facilities.

REFERENCES

[BOEHM 81] BARRY B BOEHM - SOFTWARE ENGINEERING ECONOMICS. Prentice - Hall, Inc.

[ITI 85] Specification Globales du Système de Développement d'avionique
ITI 185/84 ed. 2 du 5 Novembre 1984.

[BRANDSTAD 80] Martha A. BRANDSTAD, JOHN C. CHERNIAVSKY, National Bureau of standards, W. RICHARDS ADRION, National Science Foundation, IN COMPUTER, December 1980.

[ROYCE 70] WW. ROYCE, "MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS : CONCEPT AND TECHNIQUES", Proceeding, WESCON, August 1970.

[TURING 50] A.M. TURING "CHECKING A LARGE ROUTINE". Report of a conference on high speed automatic calculating machines. University Mathematical Laboratory, CAMBRIDGE, January 1950.

Cycle de vie d'un Système

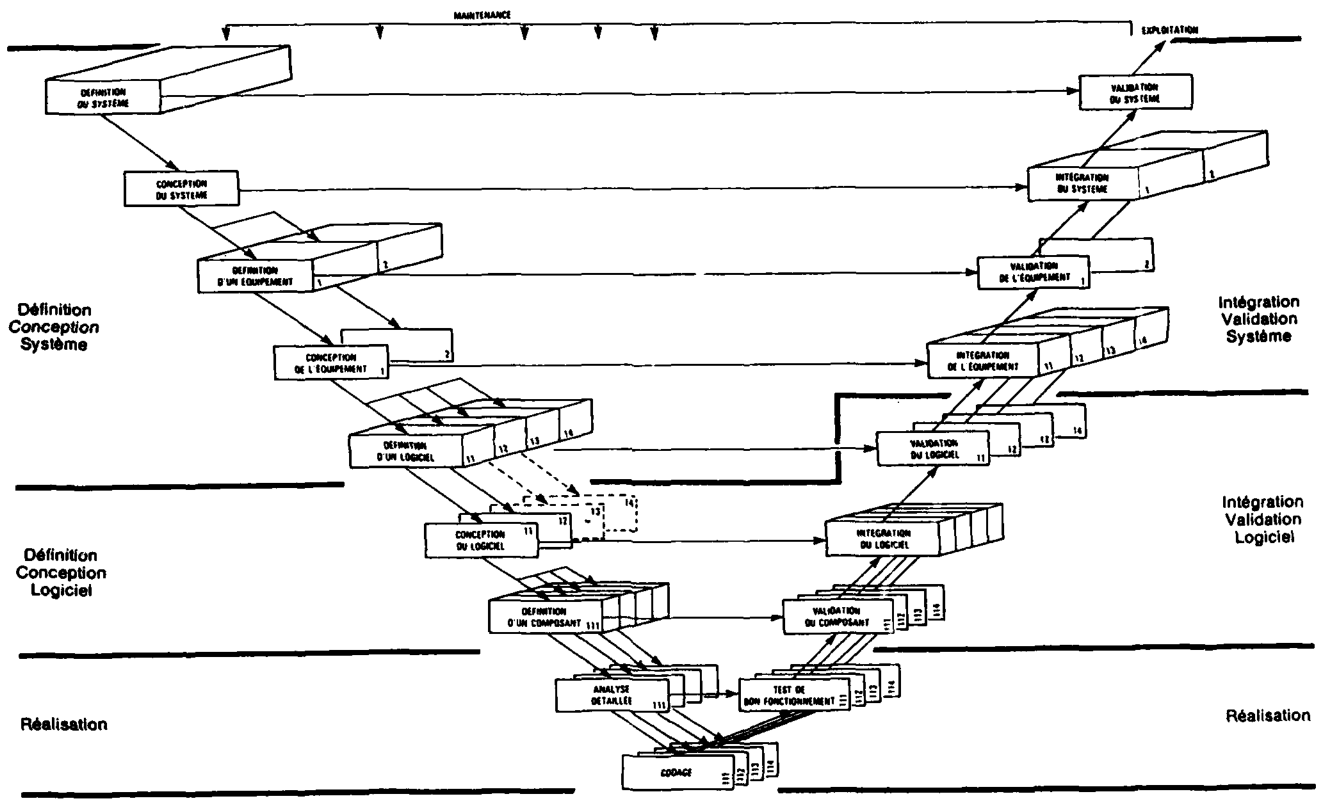

MAINTENANCE
EXPLOITATION
DÉFINITION DU SYSTÈME
CONCEPTION DU SYSTÈME
DÉFINITION D'UN ÉQUIPEMENT
CONCEPTION DE L'ÉQUIPEMENT
DÉFINITION D'UN LOGICIEL
CONCEPTION DU LOGICIEL
DÉFINITION D'UN COMPOSANT
ANALYSE DÉTAILLÉE
CODAGE
TEST DE BON FONCTIONNEMENT
VALIDATION DU COMPOSANT
INTÉGRATION DU LOGICIEL
VALIDATION DU LOGICIEL
INTÉGRATION DE L'ÉQUIPEMENT
VALIDATION DE L'ÉQUIPEMENT
INTÉGRATION DU SYSTÈME
VALIDATION DU SYSTÈME
Définition Conception Système
Définition Conception Logiciel
Réalisation
Intégration Validation Système
Intégration Validation Logiciel
Réalisation

Figure 1

DISCUSSION

P.R.Tillman, UK

I submit that the reasons many users want to change software is because many systems are so inflexible that that is the only way to make changes. Should we not design more flexible systems that can be operationally reconfigured without changing the software?

Author's Reply

Systems may be designed for flexibility only as far as we are able to foresee them. Unexpected requirements will always cause software modifications. However, flexibility should be considered a quality factor.

W.Urschel, US

Current aircraft systems have relatively few firmware processors, as the majority of vehicle control systems are analog or hydro-mechanical. However, future aircraft promise to have all vehicle control systems implemented with digital controls and the total sum of these processors may outnumber the operational software processors. How do you propose to support the firmware software with this scenario?

Author's Reply

With the current state of the art, the recommendation is to have the original developer accomplish the modification of the firmware systems.

M.Jacobsen, Ge

Can you comment on the cost/efforts related to software know-how transfer to another party in relation to the original development cost?

Author's Reply

The relative costs of transfer vary for each program because they depend on exactly what is transferred and on what commercial aspects are involved. Generally the transfer process implies some form of re-development. The efforts associated with this re-development are of the same order of magnitude as those associated with the initial development.

MAITRISE DES LOGICIELS EMBARQUES

par

Monique Slissa et Pierre Villedieu
Aérospatiale — Division Hélicoptères
13725 Marignane Cedex
France

1. INTRODUCTION :

Les conditions industrielles actuelles, associées à l'évolution des systèmes embarqués, font que le logiciel prend une place sans cesse plus importante, voire critique, dans le développement d'un système avionique.

C'est pourquoi, l'avionneur, Maître-d'Oeuvre du système, se doit de prendre en compte le développement des logiciels, en assurant la responsabilité ainsi qu'une part des activités de réalisation.

L'objet de cette communication est de présenter plus en détail :

- les conditions industrielles,
- la stratégie générale du maître-d'oeuvre système,
- la politique de développement du logiciel,
- le développement de logiciels embarqués à l'Aérospatiale DH.

2. EVOLUTION DU CONTEXTE INDUSTRIEL :

2.1. Complexité des systèmes actuels

Les missions à remplir par un système sont de plus en plus lourdes et complexes (polyvalence des appareils, stratégies de plus en plus évoluées...). Il s'ensuit donc une complexification importante des fonctions opérationnelles que le système doit réaliser.

Ceci conduisant à un niveau d'intégration de l'information de plus en plus élevé, ainsi qu'à une gestion de mission de plus en plus complexe nécessitant une intégration croissante des commandes et visualisation.

Il est évident que ces points concourent à accroître toujours plus fortement les besoins en traitements et par là même en logiciel.

2.2. Conséquence sur les développements :

Par ailleurs, si les besoins opérationnels augmentent, les budgets impartis ne suivent pas cette inflation. De même, les séries restent peu importantes en raison des coûts importants et des besoins toujours particuliers, particulièrement dans le domaine des hélicoptères.

Ceci se traduit par la nécessité de devoir personnaliser, à chaque demande, les développements précédents.

D'autre part, ainsi qu'il a été mentionné au § 2.1., les logiciels prennent de plus en plus d'importance, il s'ensuit un accroissement des coûts relatifs entre matériel et logiciel, qu'il faut donc amortir sur des séries faibles (voir figure 1).

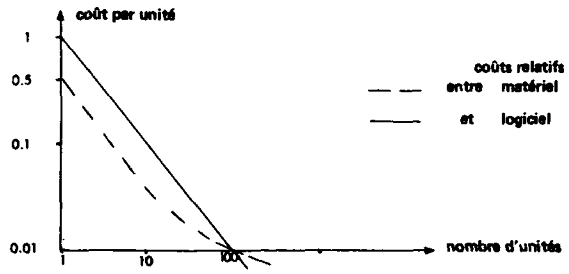


FIGURE 1

On voit donc que pour des séries peu importantes (< 100), le poids des coûts logiciels, sur une unité, est très important.

Il est alors nécessaire, lors d'une nouvelle application, de pouvoir utiliser au maximum les travaux des développements antérieurs.

Ceci offre, d'autre part, la possibilité de réduire les délais de développement d'une nouvelle application et augmente donc la *compétition de l'avionneur.*

Organiser la rentabilité des travaux de développement devient donc un objectif majeur.

2.3. Conséquence pour le Maître-d'Oeuvre :

En résumé, on voit donc que le Maître-d'Oeuvre doit être maître des coûts et des délais du développement, non seulement pour l'application en cours, mais également à plus long terme, en prévision des versions dérivées ultérieures, pour lesquelles il devra être encore compétitif.

Il doit donc maîtriser l'ensemble du système, et plus particulièrement les logiciels qui supportent la plus grande part des évolutions du système.

Cela lui impose donc de maîtriser le développement des logiciels prenant part au système sur les aspects suivants :

- Procédures et méthodes de développement, afin de pouvoir assurer les aspects "certification" des logiciels.
- Possession des droits d'usage et de modification illimités sur les produits logiciels.
- Gestion de la configuration des logiciels afin de pouvoir maîtriser les évolutions et modifications.

Cette stratégie nouvelle de la part du Maître-d'Oeuvre, en matière de logiciels, conduit à l'évidence à l'instauration de nouvelles relations entre l'avionneur et ses sous-contractants. Ceci est présenté plus en détail dans les paragraphes suivants, et plus particulièrement § 3 et § 5.3.

3. STRATEGIE GENERALE :

Il convient tout d'abord, de préciser quels sont les fondements et les bases de la stratégie générale en matière de logiciels.

Ceux-ci trouvent leurs principes dans :

- les liens entre les travaux de compétence "système" et les travaux "logiciels",
- les caractéristiques particulières des différentes fonctions opérationnelles à traiter.

3.1. Processus de développement système/logiciel :

La spécification d'un système et de ses constituants est établie par un processus de définition descendante. Elle est enrichie à chaque étape et les évaluations de complétude qui peuvent être faites permettent de s'assurer pas à pas de sa cohérence.

Dans un premier temps, la **phase de prédéfinition** a pour but l'établissement des spécifications générales du système sur la base du concept choisi par le Maître-d'Oeuvre pour répondre aux exigences opérationnelles issues des besoins exprimés du client, au travers des missions à accomplir.

Les spécifications techniques du système, des sous-systèmes d'équipements y participant sont établies au cours de **la phase de définition** proprement dite, en prenant en compte l'ensemble des contraintes imposées par le contrôle industriel, les clients...

Ces travaux conduisent à :

- la détermination de l'architecture fonctionnelle du système en décomposant les fonctions opérationnelles précédemment recencées jusqu'au niveau "élémentaire" des fonctions dites "système" pouvant être entièrement mises en oeuvre dans un équipement.
- La superposition de l'architecture fonctionnelle sur l'architecture matérielle du système.
- La caractérisation du système en ce qui concerne la fiabilité, la sécurité, la maintenabilité...

La définition acquise est ensuite détaillée. Les exigences logicielles relatives aux fonctions "système" étant établies, le travail des équipes logicielles commence. Comme le montre la figure 2, la conception préliminaire du logiciel des équipements que l'avionneur *veut maîtriser est faite à partir de l'analyse des exigences logicielles. Pendant ces travaux, les contraintes suivantes induites par le* logiciel sont émises par les équipes "système" qui poursuivent en parallèle la définition détaillée (tailles mémoire et puissance de calcul requises par l'implantation des fonctions logicielles, mécanismes temps-réel, contraintes spécifiques,...).

Celle-ci doit être terminée quand les logiciels passent dans la phase de réalisation.

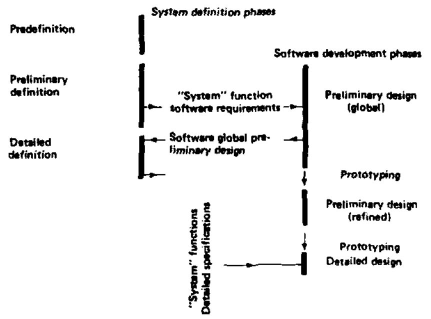

System definition phases
Predefinition
Software development phases
Preliminary definition
"System" function software requirements
Preliminary design (global)
Detailed definition
Software global preliminary design
Prototyping
Preliminary design (refined)
"System" functions Detailed specifications
Prototyping
Detailed design

FIGURE 2

3.2 Logiciels d'application :

On a vu précédemment que les évolutions entre deux systèmes portaient essentiellement sur le logiciel, pour être plus complet, on peut dire qu'un système est dérivé d'un autre par :

- adjoint ou retrait d'équipements simples, réalisant une seule fonctionnalité (senseur,...) - équipements "mono-fonction" -
- modification des logiciels des calculateurs principaux, centralisant plusieurs fonctions (calculateurs de gestion,...) - équipements "multi-fonctions" -

Il apparaît donc deux types de fonctions opérationnelles :

- Fonction **indépendantes**, implantées dans les équipements mono-fonction.
- Fonctions **liées**, implantées dans les équipements multi-fonctions.

Chacun de ces types de fonctions possède des caractéristiques particulières, en fonction desquelles la stratégie de développement du Maître-d'Oeuvre, sera modulée.

Fonctions indépendantes :

Leur but principal est d'élaborer un ensemble d'informations.

Elles sont caractéristisées par les points suivants :

- Stabilité dans le temps. Leurs spécifications évoluent peu d'une application à l'autre ;
- Nécessitent un savoir-faire spécifique à un domaine particulier.

Fonctions liées :

Elles utilisent des informations en provenance de l'ensemble du système.

Elles sont donc caractérisées par :

- Nécessité d'adaptation d'un système à un autre,
- évolutions de leurs spécifications même au cours d'un développement, en fonction des exigences clients, d'analyses opérationnelles complémentaires,...
- contiennent le savoir-faire spécifique de l'avionneur ainsi que des
- éléments industriellement confidentiels.

En conséquence de ces caractéristiques, on peut décomposer ces fonctions en deux groupes :

- Les fonctions liées "sensibles", par rapport à la stratégie industrielle de l'avionneur ;
- Les fonctions liées "non-sensibles".

Le Maître-d'Oeuvre doit donc parfaitement maîtriser le développement des logiciels des équipements multi-fonctions, au travers d'une politique de développement de logiciel stricte, mais tient compte des différences de "sensibilité" entre les fonctions.

4. POLITIQUE DE DEVELOPPEMENT DE LOGICIEL :

4.1. Objectifs du Maître-d'Oeuvre :

Les objectifs que l'avionneur fixe à sa politique en matière de logiciel sont, bien évidemment :

- La réduction des coûts et des délais,
- l'augmentation de la qualité des produits logiciels.

Pour les atteindre, il faut mettre en place les procédures et les moyens permettant d'accomplir les actions suivantes :

- Réutilisation des produits logiciels,
- Minimisation des risques d'erreur par :
  - . minimisation du nombre de modifications et reprises en cours de développement,
  - . minimisation du nombre d'erreurs résiduelles après mise en service.
- Maîtrise de la définition des produits logiciels.

Ces actions sont mises en pratique au travers d'une méthodologie de développement. Les interdépendances entre ces différentes notions sont schématisées figure 2, pour former un concept de "productivité logicielle".

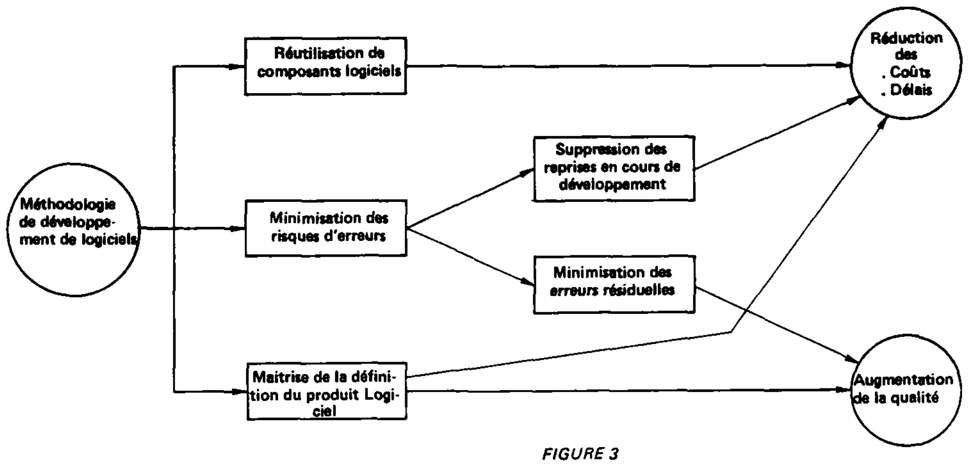

Réutilisation de composants logiciels
Réduction des . Coûts . Délais
Suppression des reprises en cours de développement
Méthodologie de développement de logiciels
Minimisation des risques d'erreurs
Minimisation des erreurs résiduelles
Maitrise de la définition du produit Logiciel
Augmentation de la qualité

FIGURE 3

4.2. Concept de productivité :

4.2.1. Réutilisabilité :

Traditionnellement, la réutilisation de logiciels est synonyme de récupération de code source.

Cette idée est trop restrictive, la réutilisation de logiciels pour être efficace, doit être vue comme la récupération de produits issus de chaque phase du développement.

Ainsi, selon l'importance des évolutions demandées, le taux de récupération des produits logiciels d'un développement précédent pourra être adapté, l'objectif étant d'avoir, quelles que soient les évolutions nécessaires, des produits récupérables.

Ceci impose des contraintes qui doivent être prises en compte dans le processus de développement logiciel :

- Définition de phases clairement identifiées à l'intérieur du cycle de développement ;
- Définition de composants logiciels, représentant les états successifs d'un logiciel d'application à chaque phase du développement ;
- Définition des produits logiciels associés à ces composants ;
- Standardisation des méthodes et des procédures de développement.

En se basant sur des modélisations classiques de développements logiciels (volume de travail par phase,...) l'estimation des gains dûs à la réutilisation des produits logiciels est donnée figure 4.

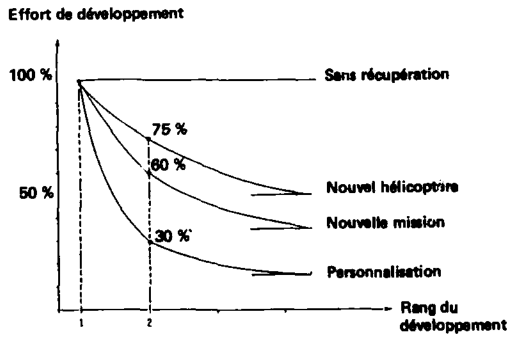


FIGURE 4

Ainsi, en résumé, la méthodologie de développement de logiciel intègre les règles de réutilisabilité qui permettent une récupération optimale des composants et produits logiciels, ainsi qu'il est schématisé figure 5.

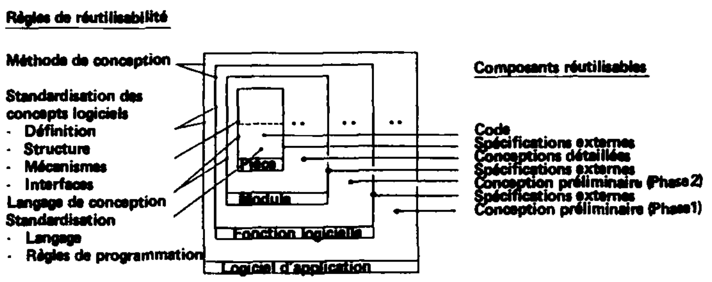


FIGURE 5

Par ailleurs, il est à noter que cette approche méthodologique doit être supportée par des outils adaptés, intégrés dans un atelier logiciel - voir paragraphe 5 -

4.2.2 <u>Minimisation des risques d'erreurs</u> :

Les sur-coûts engendrés par une erreur lors du développement d'un logiciel, dépendent, à l'évidence, du moment où cette erreur est diagnostiquée et isolée.

Ainsi, en partant, comme précédemment au § 4.2.1, d'une modélisation classique d'un développement logiciel, on est capable d'évaluer les sur-coûts engendrés par une erreur, selon qu'elle est isolée plus ou moins tardivement. Ceci est présenté figure 5.

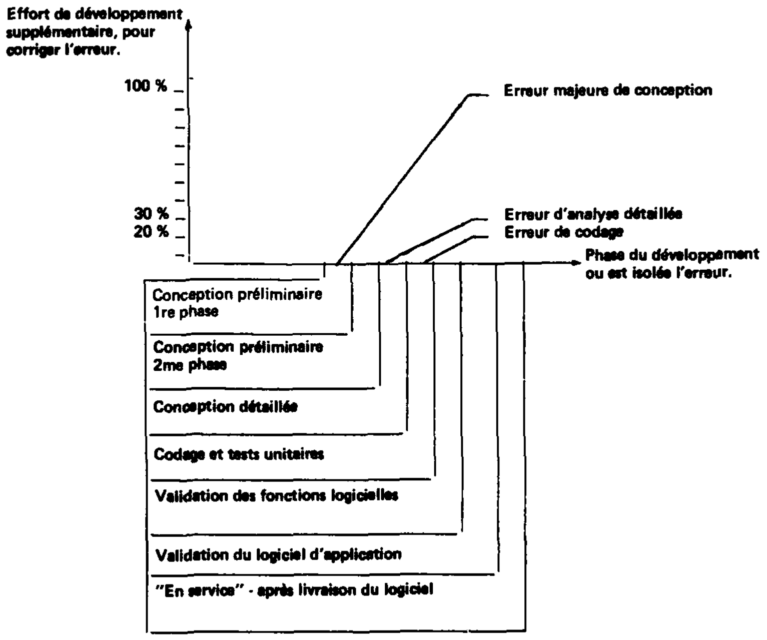


FIGURE 6

Il est donc capital d'introduire les contraintes de développement, imposées par les actions exposées dans les paragraphes précédents pour minimiser les risques d'erreurs, dans la méthodologie de développement.

<u>Ces contraintes sont les suivantes</u> :

- Nécessité de valider les résultats de chaque phase.
- Nécessité de vérifier la cohérence des produits à chaque phase.

- Nécessité d'isoler les composants logiciels (à tous les niveaux) en standardisant leurs frontières - interfaces et règles de visibilité -
- Edicter des règles d'utilisation de l'Atelier-Logiciel de la méthodologie et mettre en oeuvre les structures de projet permettant de les utiliser.

4.2.3. Maîtrise de la définition du logiciel :

La définition du logiciel comporte deux aspects :

- Aspect **technique** recouvrant les problèmes liés aux caractéristiques du logiciel.
- Aspect **maîtrise d'oeuvre système** qui répercute sur le logiciel les problèmes, décrits précédemment, et découlant des contraintes imposées à l'avionneur.

Aspect technique :

Son objectif est de maîtriser la définition du logiciel afin d'en contrôler les coûts et délais de développement.

Ceci impose :

- récupération de l'expérience acquise lors des développements antérieurs afin de faire une estimation correcte des coûts,
- préparation d'un modèle fiable du développement, adapté aux particularités de chaque application, afin de pouvoir *suivre avec précision*, l'évolution des coûts réels par rapport aux estimations initiales en fonction des phases du développement.

Aspect Maître-d'Oeuvre système :

Comme on l'a vu précédemment, § 3.2., l'avionneur, Maître-d'Oeuvre système doit, pour répondre aux contraintes nouvelles qui lui sont apparues :

- prendre en charge la responsabilité du développement des logiciels,
- participer à ce développement (voir § 5.3),
- maîtriser la gestion de configuration des logiciels.

Ces aspects de la maîtrise de la définition du logiciel sont pris en compte par l'introduction, dans l'Atelier-Logiciel, d'outils spécifiques (voir § 5.2).

5. DEVELOPPEMENT DE LOGICIEL PAR LE MAITRE-D'OEUVRE

5.1. Méthodologie Aérospatiale - Division Hélicoptères :

La méthodologie suivie par l'Aérospatiale - DH dans ses développements de logiciels embarqués repose sur trois points que l'on définira, un peu plus en détail ci-après :

- Un cycle de développement.
- Des activitées clairement définies pour chaque phase.
- Une méthode de développement, regroupant la philosophie de travail ainsi que les règles d'utilisation des outils de l'Atelier-Logiciel qui en découlent.

5.1.1. Cycle de développement :

Le cycle de développement logiciel utilisé suit les exigences établis par les normes et standards en vigueur et plus particulièrement la MIL - STD - 2167A.

Ce cycle est schématisé figure 7.

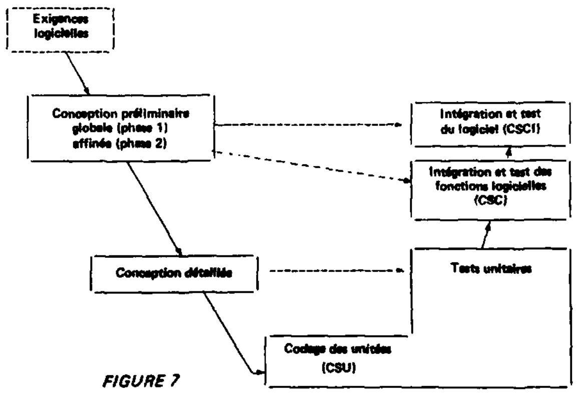

Exigences logicielles
Conception préliminaire globale (phase 1) affinée (phase 2)
Intégration et test du logiciel (CSCI)
Intégration et test des fonctions logicielles (CSC)
Conception détaillée
Tests unitaires
Codage des unités (CSU)

FIGURE 7

5.1.2. Activités des phases :

On ne donne ici qu'une description brève, définissant uniquement le type de travaux réalisés.

Exigences logicielles :

L'analyse de la spécification d'exigences logicielles, fournie par l'équipe chargée de l'étude du système, est faite par les spécialistes "logiciel" de façon à exprimer ces spécifications système d'un point de vue logiciel, et pour ajouter les exigences logicielles spécifiques.

Conception préliminaire globale :

Ceci est le premier niveau de la phase de conception globale. Il conduit à la définition des fonction logicielles - au CSC - de l'application. Au terme de cette étape, les spécifications externes de chaque fonction logicielle sont produites, de façon à transférer les activités vers l'étape suivante.

Les résultats de cette phase sont prototypés, de façon à valider leur cohérence sans les aspects temps réel et interfaces entre fonctions.

Conception préliminaire affinée :

Cette étape peut être considérée comme la conception préliminaire de chaque fonction logicielle, basée sur les spécifications externes issues de l'étape précédente. Ceci conduit à la spécification externe des modules définissant une fonction logicielle.

Chaque fonction est prototypée, à partir des résultats de cette étape, pour en valider leur cohérence.

Evidemment, le nombre de niveaux de raffinement peut être différent d'une application à une autre, en fonction du niveau de criticité de l'application.

Conception détaillée :

Chaque module logiciel est analysé en détail, sans l'aspect programmation, à partir de ses spécifications externes. Ceci fournit donc une spécification d'analyse détaillée de chaque module, englobant la définition des unités de programmation - ou CSU -

Codage :

Lors de cette phase le code source des unités est écrit et compilé, à partir des spécifications d'analyse détaillée.

Tests unitaires :

Chaque unité (CSU) est testée indépendamment, d'après les spécifications de tests unitaires incluses dans la spécification d'analyse détaillée du module.

Intégration et test des Fonctions Logicielles :

Les modules logiciels, formant une fonction logicielle, sont intégrés et la fonction testée d'après les spécifications de tests écrites lors de la conception préliminaire (deuxième phase).

On aura donc autant de niveaux d'intégration que de niveaux de conception préliminaire.

Intégration et test du logiciel :

Les fonctions logicielles, constituant le logiciel d'application d'un équipement (CSCI), sont intégrées et ce logiciel est alors testé en regard de ses spécifications de test écrites lors de la conception préliminaire (première phase).

5.1.3. Méthode de développement :

La méthode appliquée lors d'un développement de logiciel est basée sur l'expérience acquise lors de précédents développements. En conséquence, on donne ci-dessous les bases de la méthode appliquée par l'Aérospatiale - DH :

- Deux points de vue différents doivent être adoptés pour les deux phases de conception - préliminaire et détaillée de façon à :

  . Introduire toutes les fonctionnalités du logiciel ainsi que leurs inter relations (Temps réel et interface) lors de la conception préliminaire :

  Ceci constitue le point de vue "fonctionnel" qui doit être adopté pour la conception préliminaire.

  . Séparer les exigences logicielles des contraintes de réalisation :

  C'est donc le point de vue "réalisation" qui doit être adopté pour la conception détaillée.

- Les activités de chaque phase d'un développement doivent être strictement remplies, les résultats d'une activité sont nécessaires pour exécuter correctement l'activité suivante :

- De façon à minimiser les erreurs et les modifications au cours d'un développement et à assurer le plus haut niveau de fiabilité pour le logiciel, les résultats des activités doivent être validés avant qu'ils soient utilisés pour la phase suivante.

Définition de la méthode :

La description de la méthode est donnée ci-dessous (figure 8).

PHASE	METHODE	COMMENTAIRES
Conception préliminaire	Modélisation descendante et hiérarchisée du logiciel, intégrant les mécanismes temps-réel.	Modélisation du logiciel en introduisant simultanément les aspects temps-réel et fonctionnels, au travers d'un processus de raffinement descendant.
Prototypage	Implémentation ADA des modèles du logiciel avec vérification de cohérence.	Validation des modèles du logiciel en exécutant, sur calculateur hôte, cette implémentation ADA.
Conception détaillée	Utilisation du langage de haut niveau comme langage de conception.	Conversion du modèle de conception en un modèle de réalisation.
Codage	Traduction du modèle de réalisation en instructions du langage de haut niveau.	Raffinement du modèle de réalisation.

FIGURE 8

Outils supports de cette méthodologie :

Il est évident que des outils sont nécessaires pour supporter la méthode du développement décrite ci-dessous.

Ces outils doivent donc satisfaire aux exigences suivantes :

- Les outils devant supporter des points de vue différents doivent être différents.

  Donc les outils supportant la conception préliminaire seront différents des outils supportant la conception détaillée.

- ADA est le langage de haut niveau retenu par l'Aérospatiale - DH.

Les outils utilisés dans l'Atelier-Logiciel Aérospatiale - DH sont décrits dans le paragraphe suivant.

5.2. Atelier-Logiciel Aérospatiale - DH :

On décrit ici, brièvement, les outils mis en place et intégrés dans un atelier logiciel, par Aérospatiale - DH pour supporter sa méthodologie de développement de logiciels embarqués.

Exigences logicielles :

L'outil EPOS-R est utilisé pour décrire les exigences logicielles en introduidant dans le texte des mots clés pouvant être repris et testés lors de la conception préliminaire.

Conception préliminaire :

EPOS-R supporte les deux étapes de la conception préliminaire selon une approche hiérarchisée descendante.

Grâce à cet outil, le logiciel est structuré simultanément sous les deux aspects : **fonctionnel et dynamique**.

Prototypage :

Les modèles issus des phases de conception préliminaire sont traduits en ADA, compilés et exécutés pour valider l'architecture logicielle.

Conception détaillée :

L'analyse détaillée des modules, issus de la conception préliminaire, est faite à l'aide du langage ADA, utilisé comme un langage de conception.

Toutes les étapes de cette conception détaillée sont compilées et donc validées.

Codage :

Les nouveaux développements de logiciels embarqués à Aérospatiale - DH sont fait avec ADA comme langage d'implémentation. Jusqu'à présent d'autres langages ont été utilisés, dans le cadre de cet atelier logiciel (Pascal LTR V2, C, FORTRAN,...).

Gestion de configuration :

La gestion de configuration de production (sources, objets,...) est faite à l'aide de l'outil PALAS-B.

Actuellement, la gestion de configuration complète - Structure d'Accueil gérant tous les objets associés à un logiciel, est faite également par PALAS-B. Cependant, fin 88, cette Structure d'Accueil sera faite à l'aide d'un outil spécifique construit autour d'une base de donnée relationnelle ORACLE.

<u>Outils complémentaires</u> :

La gestion de l'expérience en matière de logiciels est faite à l'aide d'un outil, développé par Aérospatiale - DH, appelé OML, permettant de faire les estimations de coûts d'un développement en fonction de :

- l'influence de divers critères (complexité, type d'application,...),
- les caractéristiques et les résultats mémorisés et ordonnés lors des développements antérieurs.

Le suivi des projets logiciels est, alors, fait à l'aide de EPOS-P, à partir des données initiales d'OML-A chaque phase du développement, les données OML sont mises à jour en fonction des résultats, problèmes remontés,...).

L'atelier logiciel Aérospatiale - DH est schématisé ci-dessous (figure 9).

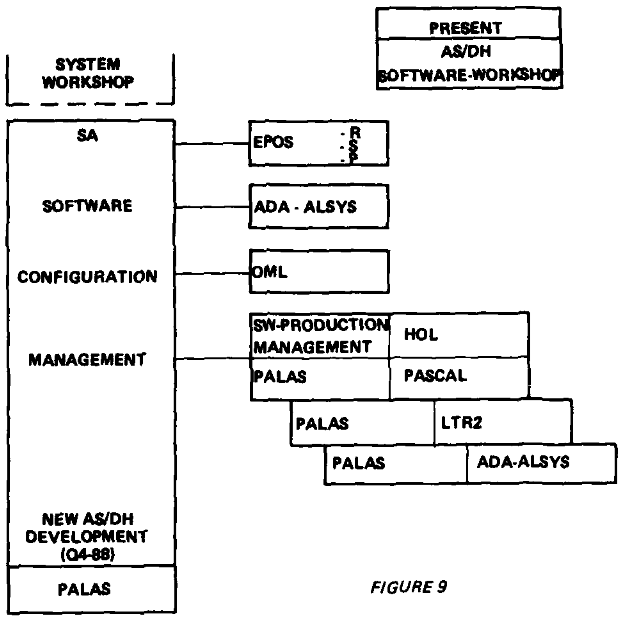


FIGURE 9

5.3. <u>Relations Maître-d'Oeuvre / Sous-Contractants</u>

Comme il a été dit précédemment, l'application de cette stratégie en matière de logiciels, nécessite la mise en place de relations, entre l'avionneur Maître-d'Oeuvre et ses sous-contractants, adaptées à cette situation.

Ces relations entre l'Aérospatiale - DH et ses sous-contractants sont décrites dans le tableau ci-dessous (figure 10).

	RESPONSABILITE DES LOGICIELS	DEVELOPPEMENT DES LOGICIELS	DEFINITION DES PROCEDURES DE DEVELOPPEMENT
Développement d'applications expérimentales	Aérospatiale - DH	Aérospatiale - DH	Aérospatiale - DH
Calculateurs de gestion système (multi-fonctions)	Aérospatiale - DH	Conception préliminaire: Aérospatiale DH; Fonctions sensibles: Aérosp. DH; Non sensibles: Sous-contrac.	Aérospatiale - DH
Equipements mono-fonction	Sous-contractant	Sous-contractant	Sous-contractant (selon des exigences Aérospatiale - DH).

FIGURE 10

L'enchaînement des activités prises en compte par le Maître-d'Oeuvre en fonction des phases du développement complet est décrit figure 11.

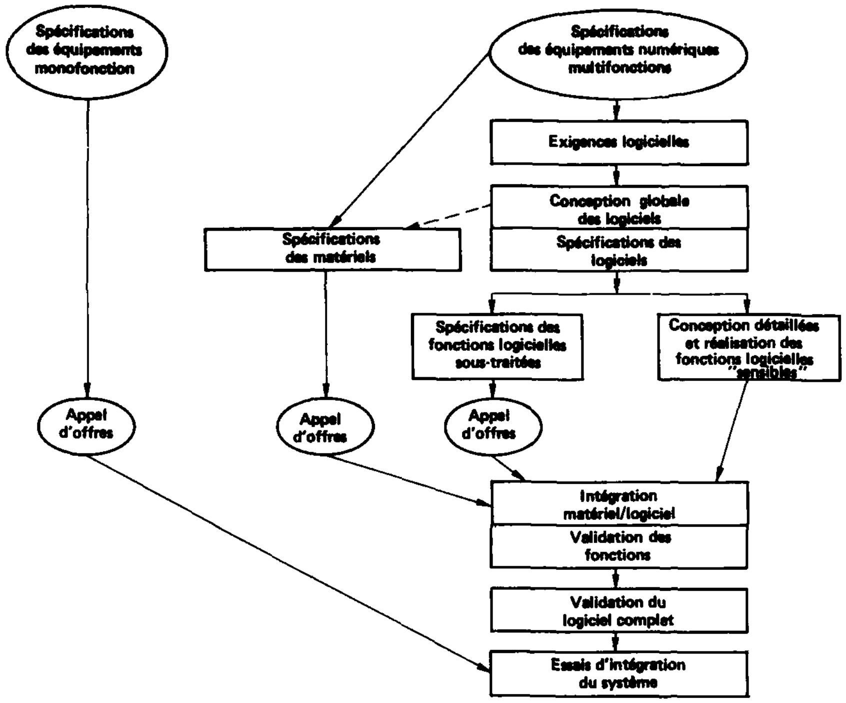

Spécifications des équipements monofonction
Spécifications des équipements numériques multifonctions
Exigences logicielles
Conception globale des logiciels
Spécifications des logiciels
Spécifications des matériels
Spécifications des fonctions logicielles sous-traitées
Conception détaillées et réalisation des fonctions logicielles "sensibles"
Appel d'offres
Appel d'offres
Appel d'offres
Intégration matériel/logiciel
Validation des fonctions
Validation du logiciel complet
Essais d'intégration du système

FIGURE 11

6. <u>CONCLUSIONS</u> :

La stratégie qui a été décrite ci-dessus est aujourd'hui appliquée dans les programmes d'avionique en cours, à la Division Hélicoptères de l'Aérospatiale. L'ensemble des moyens et des structures nécessaires à son impact, dans le cadre des grands programmes nouveaux est en place. Les évolutions futures auront pour but de maintenir leur adéquation aux fins recherchées et principalement 'a maîtrise des coûts et des délais.

CONVERSION TO ADA: DOES IT REALLY MAKE SENSE?

Robert A. Converse
Mitchell J. Bassman

Computer Sciences Corporation
6565 Arlington Boulevard
Falls Church, Virginia 22046 USA

SUMMARY

Change is an integral part of any useful operational system. Changes are required for any number of reasons, ranging from minor errors that exist in the system to major system upgrades to meet totally new and different requirements.

For the U. S. Department of Defense (DoD), the manner in which systems are changed is of significant interest. Major system upgrades occur in virtually every system at various times during their operational lifetime. One way to manage the changes and to reduce the long term costs associated with them is the use of the Ada programming language [1].

Avionics is an application area within DoD for which the use of Ada is a serious consideration. However, in addition to reducing the long term costs, the avionics software must also meet stringent real-time performance and resource utilization requirements.

This paper addresses some of the issues associated with the use of Ada to accomplish system changes. Background and general issues will be discussed as well as some concerns that are specific to avionics systems.

INTRODUCTION

Ada was developed by the DoD to provide a standard language for the development and maintenance of mission-critical systems. Initially standardized by DoD in late 1980, Ada has been steadily maturing both as a language for large system development and as the basis for a comprehensive set of tools that support large system development and maintenance. Ada is ready. Production quality Ada compilers are available for most major computers available on the market today. The processors supported range from microprocessors such as the Intel 80x86 and Motorola M680x0 families through minicomputers such as the DEC VAX family to the very large mainframes such as those produced by IBM, UNISYS, and CDC. Software engineering tools that support software development methods designed to take advantage of the advanced features of the Ada language are readily available.

Ada is mature and ready. Why should it be considered as a potential language for the upgrade of a system? For the DoD community, there are two reasons: DoD policy and technical advantages.

From a policy standpoint, Ada was developed by the DoD to solve the problem of the lack of commonality among major DoD systems. Implementation in a common language would provide a basis for reusability and transportability of software components among systems throughout the DoD community. This capability would lead to reduced costs for system development and system maintenance as well as increased availability of people who know the language, tools, and methods used to develop those systems. With this goal in mind and with a mature language at its disposal, DoD has issued policy statements requiring the use of Ada for all new starts and for all major system upgrades for mission-critical systems. Further, the use of Ada for non-mission-critical systems is strongly encouraged [2,3].

Ada is mature, ready, and is required. Why is Ada "better" than other languages such as FORTRAN and COBOL? What is the technical advantage to be gained by using Ada? Ada, while still a third generation (procedural) language, was designed to support design and development at a higher level of abstraction than other procedural languages. However, no single feature distinguishes Ada from other third generation languages. Rather, the discriminating factor is its collection of features in a single language. Ada's key features are strong typing, abstraction mechanisms, packages, tasks, generics, and exceptions. Ada's type constructs and abstraction mechanisms require that interfaces between software components be clearly and completely specified and that the compiler (and run-time system) enforces those specifications. Ada facilitates the partitioning of a large complex system into a number of smaller and more manageable components through the use of the package and task constructs. Generic units permit the construction of a template for an algorithm that can be used with the types and abstractions previously defined, thereby making it unnecessary to write a different program for each type. Exceptions permit the developer to create specific error handlers for unanticipated problems. Features such as these allow programs written in Ada to be designed for change; the changes can be isolated, and the impact of a change can be easily determined.

UPGRADE DISCUSSION

Planning for a major system upgrade requires considering several factors. The more important factors include:

- o the expected operational lifetime of the system
- o the degree of modularity within the system
- o the level of integration associated with the upgrade
- o the performance and resources.

Operational Lifetime Factor

The expected operational lifetime of a system directly impacts the expected return on investment associated with the cost of a change. If additional major system upgrades are expected for the system, then planning for subsequent change during an upgrade makes sense. However, if no additional upgrades are anticipated, e.g., a replacement system is already in development, the cost for the upgrade should be minimized. Ada is designed to provide significantly lower costs during a system's operational lifetime. It provides a mechanism for incorporating change, reducing the costs of a controlled evolution of the system in subsequent upgrades.

Modularity Factor

The modularity factor, the degree to which the upgrade affects the entire system, affects the selection of the approach to the upgrade. For an upgrade to a tightly coupled system, the upgrade may require significant modifications to the existing software within the system as well as the development of new or additional software to meet new operational requirements.

Integration Factor

The integration factor considers the extent to which the capabilities to be provided by the upgrade are separate from the existing system capabilities. For a totally upward compatible upgrade to a system, the interface between the existing system and the new functionality may be sufficiently well-defined to allow the upgrade to be developed with no impact on the existing system.

Performance and Resources Factors

The performance factor considers the amount of processing to be accomplished to meet the required functionality of the system. A major part of this factor is the extent of real-time response necessary. The resource factor considers the physical limitations that may be imposed on the system. An example of such a limitation is the amount of memory available for a program.

Accomplishing an upgrade requires considering two alternatives:

(1) use of some language other than Ada or

(2) transition to the use of Ada.

Each of these is discussed below.

Non-Ada Alternative

A decision to use some language other than Ada for a system upgrade is based on the factors described above as well as the applicable policy statements. For a system with a short operational lifetime, it may be cost-effective to implement the upgrade in the language in which the system was originally developed. Of course such a decision assumes that the target computer has sufficient capacity to accommodate the upgraded system. If the current target computer lacks that capacity and a new environment must be acquired, then concurrent transition to the Ada language may be the proper choice.

Ada Alternative

Implementing a major system upgrade in Ada can be accomplished by three general approaches:

(1) redesigning and reimplementing the entire system in Ada,

(2) implementing the upgrade in Ada and interfacing the upgrade to the existing system, or

(3) converting the existing software to Ada while incorporating changes that are designed for Ada.

Redesign and Reimplement in Ada. This approach, while providing the most power and flexibility, also involves the highest initial cost. However, it may actually provide significantly lower life-cycle costs for those systems that have a long operational lifetime.

An example of a system for which this is a good solution is the U.S. Navy's Submarine Satellite Information Exchange System II (SSIXS II) upgrade for the shore-based portion of the system. The existing system is implemented in the Ultra-16 assembly language for the Navy standard AN/UYK-20 computer. The system uses all of the computing power and memory available in the AN/UYK-20, leaving no additional capability for the proposed enhancements. Therefore, the Navy chose to change the target computer as a part of the major system upgrade.

The initial major system upgrade was to redesign and reimplement the existing system in Ada for execution on a DEC MicroVAX II computer and to incorporate a few engineering changes in the initial release. However, the new design was structured to facilitate the incorporation of further changes on an upgrade schedule that calls for three additional releases.

The redesign and reimplement approach is harder than doing an initial design in Ada because of the requirement to emulate the existing system interfaces to the user and to other communications systems. However, the approach is cost-effective in cases such as SSIXS II (Shore) because:

- the system has a long operational life expectancy
- planning for additional changes is included in the design
- a new target computer was defined for which Ada is available.

Interface Ada Upgrade to Existing System. Mixing Ada and non-Ada components within a system is particularly attractive for a very large system. This approach requires careful consideration of the modularity and integration factors. For an upgrade to a system that is not modular, mixing Ada and non-Ada may be impossible; the existing system may require too many changes to permit only the changes to be in Ada. In that case, either the redesign and reimplement approach or the use of the non-Ada alternative should be considered. For an upgrade for which the existing system will be enhanced with the addition of new components for which a well-defined interface exists, the use of Ada for the enhancements may be possible.

Selecting this approach depends significantly on the run-time environment for the operational system. Ada requires specific run-time support for data management, exception handling, and task management. Thus, the target system, including both the computer itself and the operating system on which the operational system will execute, must be supported by an Ada compiler.

This issue can be illustrated with two examples. The U.S. Army is updating a management information system to produce a Standard Financial System (STANFINS). This system is composed of five major components, four of which have been implemented in COBOL. The fifth component, the General Accounting package, is being implemented in Ada. The target computers are commercially available computers and operating systems such as the IBM 30xx products. Both COBOL and Ada compilers are available for those systems. The interfaces between the General Accounting package and the other STANFINS components are well-defined, and the commercially available operating systems support communications between programs written in Ada and those written in COBOL.

On the other hand, the U.S. Navy performed a series of analyses in the mid-1980's on the transition of three existing mission-critical systems to Ada: the TRIDENT submarine combat system, the AEGIS system, and the Restructured Naval Tactical Data System (RNTDS). Each of these has a different run-time executive system even though they all use (or will use) the Navy standard AN/UYK-43 computer. To provide a standard run-time system for Ada programs developed for the AN/UYK-43, the Navy is developing an Ada compiler and a corresponding run-time system. However, because of differences between the existing run-time systems and the one being developed for Ada, to transition these three systems to the use of that compiler and run-time system would require a redesign and reimplementation. Another approach would be to provide an Ada capability for each of the existing run-time systems. This approach would allow mixing (new) Ada programs with existing CMS-2 programs.

Convert to Ada and Enhance. A third approach for implementing a major system upgrade in Ada is first to convert the existing system to Ada and then to use Ada directly to implement enhancements. This approach is usually not the best alternative unless (1) the existing system is large and well-structured, (2) the enhancements can be phased in gradually, and (3) a reliable automatic conversion tool exists to support the translation from the old implementation language to Ada.

This approach may be the fastest way to produce an operational Ada language version of an existing system with some of the advantages inherent in the use of an Ada compiler. Since even the best automatic conversion tools will produce code using Ada syntax in the style of the original language, the primary short-term benefit will be the use of the Ada program library feature to guarantee consistency among separately compiled units.

Additional benefits, derived from the use of Ada's package construct and user-defined types, will come during the enhancement period. After the existing system has been translated to Ada, enhancements can be implemented either by modifying old components or by adding new ones. New components will be designed and implemented to take advantage of modern Ada language features; improvements to old components can be phased in gradually without affecting the operational system.

Never recommended is a strict manual translation from the existing source language to Ada. It is expensive, time-consuming, and prone to error. Unless an automatic conversion aid is available, mixing new Ada code with the old code in the original language may be a better alternative. Unless most of the old code can be reused with little or no modification, the first alternative, of redesigning the entire system to support the upgrades while taking advantage of the features of the Ada language, is the recommended approach [4].

Avionics Issues

Typical avionics applications take place in an environment in which weight, power, and volume for the processor are at a premium. Further, avionics systems tend to require high speed processing for decision making and control purposes in real time within an integrated environment. Therefore, operational avionics systems have normally been built around special purpose run-time executives and execute in special purpose processors. The application of Ada to avionics applications has come under particular study because of these characteristics. The generation of efficient object code, the minimization of run-time overhead for memory management and context switching, and the cyclic processing of sensor inputs are all issues of significant concern to the avionics system developer.

Specific solutions are starting to appear for avionics implementations in Ada, and Ada is being used for the development of avionics systems [5,6,7]. For example, as reported at the International Workshop on Real-Time Ada Issues in May 1987, Ada was used for approximately 88% of an Operational Flight Program developed and flown by General Dynamics, Fort Worth Division. However, this program, a rewrite of a Jovial implementation, required more memory and took slightly longer than the Jovial implementation. Thus, care must be exercised in the use of Ada for upgrading avionics systems. Ada can provide much in terms of reliability, adaptability, and reusability but it cannot be at the expense of performance.

General Issues

Planning. No matter which alternative is selected, a successful system upgrade requires careful planning. The best approach can be selected only after a detailed analysis of both the existing system and the needed upgrade. In addition, it is essential to plan for the transition of personnel and the computing environment.

Personnel. Two options immediately come to mind for developing a transition plan for acquiring Ada-literate personnel: (1) hire all new personnel with existing Ada expertise or (2) retrain current personnel. The first option can be dismissed quickly. Unless there is no current staff and all new personnel are being hired, some amount of staff retraining will be required. Further, even if there is no current staff, there simply are not enough available software engineers and managers with both experience and competence in the use of Ada to staff all Ada software development programs. Retraining experienced software personnel in the efficient use of Ada and modern software engineering practices is often easier and more economical than hiring new personnel with some "Ada training" but no experience.

Managers should be trained first, since a successful transition to Ada requires management support. Management support comes only with an appreciation for the potential long-term benefits to be derived after the short-term inconvenience of transition.

Initial programmer training should include an overview of the entire language in the context of modern software engineering. The emphasis should be on the high-level language features that support good software engineering. After the programmers become comfortable with the language structure, more advanced features and details, such as the interactions between different fixed point types, can be learned as they are needed on-the-job or in advanced seminars. Some people will inevitably be more effective Ada programmers than others, just as some are more effective in COBOL.

Computing Environment. Once the decision to use Ada has been made, selection of the proper development environment and software support tools should also be a carefully reasoned management decision. Often the decision will be at least partially driven by the existing hardware and operating system environment required for system development or operation.

Much has been written about Ada tools and programming support environments. Many mature Ada compilers are available for most popular computers and operating systems. Every month many new products are being released for integration into existing environments. The rash of product announcements at the combined International Ada Conference and Ada Expo during December 1987 indicates the level of activity to improve the way in which software is developed in Ada. Each vendor will be quick to point out that his product is the best. A comparative evaluation of many of the rapidly changing products would likely be out-of-date before this article is printed.

Unsupported productivity claims associated with the use of Fourth Generation Languages (4GLs) are highly suspect. Although some 4GLs may be useful as prototyping tools and others may be useful as partial code generation support for providing an interface to commercial software products, greater gains in productivity can be expected today from carefully designed reuse of software components. Although the world would welcome an automated tool that supports the effective use of Ada throughout the entire software development life cycle, no such tool is available today.

CONCLUSION

We have shown that transition to Ada for an operational system is a reasonable alternative. The technology to support Ada approaches exists, and the long-term benefits far outweigh the initial costs associated with such a change. The conclusion is inescapable: Ada's good--use it.

REFERENCES

1. Military Standard Ada Programming Language, Department of Defense, Ada Joint Programming Office, Washington, DC, ANSI/MIL-STD-1815A, 1983.

2. DoD Directive 3405.1, "Computer Programming Language Policy," April 2, 1987.

3. DoD Directive 3405.2, "Use of Ada in Weapon Systems," March 30, 1987.

4. Wallis, P., "Automatic Language Conversion and Its Place in the Transition to Ada," In Proceedings of the Ada International Conference, Paris, May 1985. Also Ada Letters, Vol. V, No. 2 (September, October 1985), pp. 275-284.

5. McCormick, F., "Scheduling Difficulties of Ada In the Hard Real-Time Environment," In Proceedings of the International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK, 13-15 May 1987. Also Ada Letters, Vol. VII, No. 6 (Fall 1987), pp. 49-53.

6. Phillips, S. and P. Stevenson, "The Role of Ada in Real Time Embedded Applications," Ada Letters, Vol. III, No. 4 (January, February 1984), pp. 99-111.

7. Sarkar, J. and T. Wong, "Impact of Ada Features on Real-Time Performances," In Proceedings of the International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK, May 1987. Also Ada Letters, Vol. VII, No. 6 (Fall 1987), pp. 88-92.

DISCUSSION

W.Mansel, Ge

You have mentioned in your presentation that for the conversion of software into Ada the mixing of Ada code with other programming language code was realized. Can you explain what languages have been considered and how this has been realized, since to my knowledge Ada compilers currently only allow the input of assembly code packages?

Author's Reply

To my understanding, there has been no direct intermixing of Ada code and non-Ada code. The systems to which I referred interface components written in Ada to components written in some other language through the underlying operating system or through a data base access scheme.

AVIONICS SYSTEMS ENGINEERING AND ITS RELATIONSHIP TO MISSION SOFTWARE DEVELOPMENT

by
Dr. Herbert Klenk / Dr. Helmut Rapp
Messerschmitt-Bölkow-Blohm GmbH
Abt. FE 31
Postfach 80 11 60
D8000 München 80
West Germany

Summary

The introduction of digital avionics, i.e. freely programmable embedded and distributed real time computer systems into military fighter aircraft and the accompanying transition from electromechanical to software-intensive systems are forcing the aviation industry into new approaches to systems design, development, integration and test. Avionic systems development and software engineering are becoming more and more intertwined desciplines that critically depend on each other. Software is no longer just one part of the system - it is the system. System functions and system performance are tightly coupled to real time mission software. Hence, because of the complexitiy of nowadays systems, software changes can no longer be considered easy and quick as compared to hardware changes.

The purpose of this paper is to describe the avionic systems development methodology established and used at MBB, its relationship to software development and the experiences made by the application of this development methodology and its related tools to such major programs as the development of the Electronic Combat Reconnaissance Tornado, the integration of the HARM missile into the Interdiction Strike Tornado and the Improved Combat Efficiency Program of the German F-4F Phantom.

As far as the weapon system Tornado is concerned, avionics system design, integration and test as well as system software development are trinational untertakings which include three different customers and different operational needs. Any systems development methodology has to reflect these prerequisites. Therefore this paper also addresses problems and solutions for international cooperation in systems and software development and their impact upon project management and control.

The toolset in use for system development requires further enhancements. Potential improvements of existing and requirements for additional tools are also being described.

1. Introduction

MBB has been awarded in the last few years three major system development contracts for military aircraft:

- F-4F Improved Combat Efficiency Program
- HARM missile integration into Tornado IDS
- Electronic Combat Reconnaissance Tornado (ECR)

These programs are characterised by a considerable increase of electronic systems complexity. This reflects the generally recognised trend to invest more and more in avionics systems in military aircraft (percentage of avionics to flyaway costs being some 10 % a decade ago, nowadays being some 30 % and more for future weapon systems). The computing capacity of the ECR Tornado increased considerably compared to the basic Tornado within 4 years: the number of on aircraft loadable computers from 1 to 6 and their memory capacity from 128 kwords to nearly 3000 kwords. 3 of the 6 computers are mission computers and were programmed by our company. Obviously changes of this size can only be realised with a disciplined methodology for system and software development.

There were also typical experiences with less rigid methods of the past that supported this view: missed schedules, budget overruns, unrealistic planning and hence loss of credibility. In the long run this affects the competitiveness of a company in an intolerable way.

MBB therfore introduced a more rigid methodology of designing complete aircraft systems. It had to be applicable for different programs without major changes, it had to support all phases of system development and it had to assure that the interfaces between systems engineering groups and software development groups were well defined. The methodology had to allow for iterations in specific phases, nevertheless assuring proper completion of each phase. Finally, as many phases as possible had to be supported by appropriate tools to enhance productivity.

The methodology adopted is closely related to DoD Std. 2167, especially in the software development phases, but had to be amended for some of the other phases. As far as software development is concerned, these modifications reflect the transition from software development for one single, centralised computer to that for distributed real time on-board computer systems. In the following, after having outlined the methodology, emphasis will be put on the influence of system development to software development and vice versa.

2. Generic System Development Model

In order to be independent of specific projects, a generic System Development Model has been developed. To be applicable for a specific project it had to be amended by implementation directions specific for the project. For the purpose of the subject of this paper, however, it is sufficient to deal with the generic model only.

The system development model adopted is depicted in Fig.1. The main features are

- o its generic set of steps or phases according to DoD Std. 2167
- o the definition of baselines, i.e. freeze of requirement and product standards throughout development
- o the functional partitioning of the operational software among the distributed computers
- o the strong emphasis on a disciplined integrated systems development approach.

Each of these phases/ activities will result in products (i.e. documents, equipment, software) that are being reviewed at the end of the specific phase of development. Ideally, the next phase will only be started after approval of the preceding one.

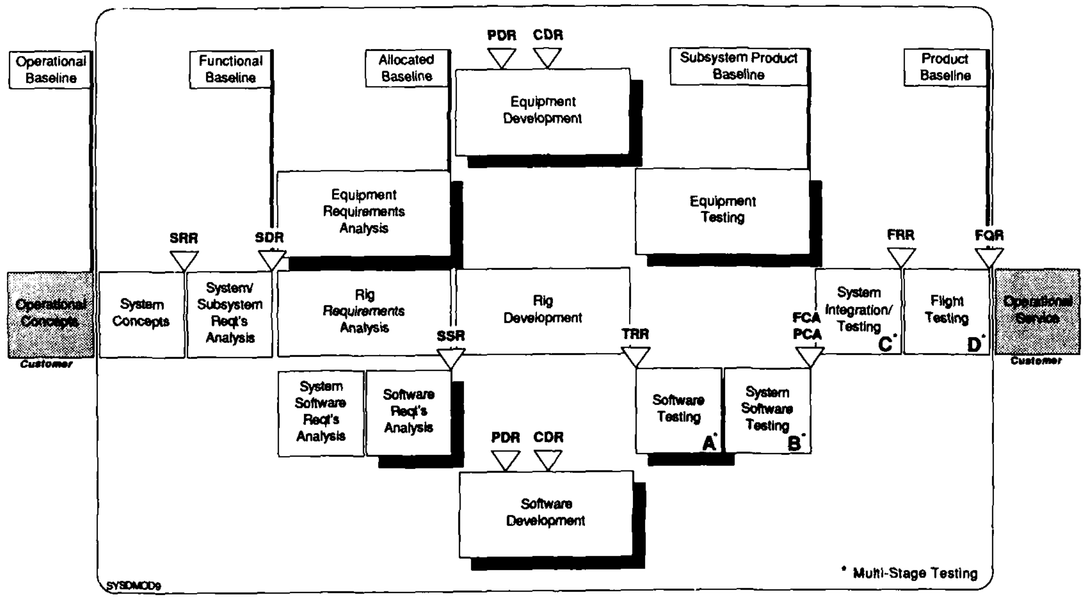


Figure 1 System Development Model

Operational Concepts

The Operational Concepts are based on the operational needs of the customer. They will be defined by the customer in close contact with the weapon system company. The Operational Concepts either define a completely new weapon system or just improvements to existing systems. They form the basis for any contract and as such establish the ***Operational Baseline.***

System Concepts

This phase will be carried out in the following steps:

- o procedural analysis of Operational Concepts
- o technical analysis of Operational Concepts resulting in a Preliminary System Concept
- o review of the Preliminary System Concept

In the first step the Operational Concepts will be analysed with respect to completeness, feasibility, consistency and impacts on related further tasks. After having clarified inconsistencies with the customer the Operational Concepts will be technically analysed resulting in a Preliminary System Concept accompanied by rough estimates for required efforts and timescales. A formal **System Requirements Review (SRR)** will be held with the customer to get the System Concept being agreed.

System/Subsystem Requirements Analysis

The Operational Concepts and the System Concept will be analysed and refined leading to the System Specification. In parallel the System Test Specification will be established. These documents will be further refined to the Subsystem Specifications, the Interface Control Document and the Subsystem Test Specifications. The **System Design Review (SDR)** finally will authorise the products of this phase thus constituting the ***Functional Baseline.*** In addition, development efforts and timescales will be revised and refined.

Equipment Requirements Analysis

In this step the functional baseline as established above will be used to derive the impact on the equipments of the system, i.e. how can the changes required for by the subsystem specification be implemented in all affected equipments. This will include the definition of the Equipment Specifications, the definition of the Equipment Test Specifications, the refinement of the ICD and will include the Equipment Specification Reviews.

Together with the results of the Software and Test Rig Requirements Analysis phase, the above will define the ***Allocated Baseline.*** At This point in time customer induced requirements must be frozen in order to proceed to full scale development in a disciplined and controlled manner.

Test Rig Requirements Analysis

Test benches on ground consisting of stimulating and recording facilities, of real aircraft hardware and of a flexible interconnection wiring we call a "test rig".

In general, new system/subsystem requirements and their corresponding test specifications will ask for enhanced test facilities (rigs) and test software. Therefore, in order to provide those in time, the test rig requirements will have to be analysed in parallel to the hardware and software requirements, comprising of the definition of the Test Rig Requirements Specification, the definition of the Test Rig Acceptance Procedure and concluding in the Test Rig Requirements Review. In addition effort and timescales will have to be defined in this step.

System Software Requirements Analysis

"System software" is defined herein as being the collection of all software packages in all freely programmable on board computers.

Based on the Subsystem Specification, the ICD and the relevant Equipment Specifications, the necessary enhancements and changes to the System Software will be analysed. This will be followed by the functional partitioning of the different software functions to specific computers. This phase will include the definition of the System Software Requirements Specification, the definition of the System Software Test Specification, the definition of the (Software) Interface Requirements Specification and will conclude in the System Software Requirements Review.

Software Requirements Analysis

Based on the System Software Requirements Specification, the ICD and the relevant Equipment Specifications, the requirements for the software of each computer will be completely defined and the impacts on all relevant Software Requirement Specifications for all relevant processors will be assessed. Thus for each computer the Software Requirement Specifications and the Interface Requirements Specifications will be established concluding in the **Software Specification Reviews (SSR).** In addition, refinements of effort and timescales have to be defined in this step.

Software Requirements Analysis in general will be carried out in parallel to Equipment Requirements Analysis. However, in case of an embedded computer it might be necessary to define the equipment first in order to be able to establish the Software Requirements Specification.

Equipment Development

Based on the Equipment Specifications the design of the equipments will be defined and the equipment will be built accordingly. For each equipment this phase will include the definition of the Equipment Top Level Design Document and the definition of the Equipment Test Plan concluding in the **Preliminary Design Review (PDR).**

After PDR authorisation the Detailed Design Document, the Interface Design Document and the Equipment Test Description will be defined concluding in the **Critical Design Review (CDR).**

Final CDR agreement will authorise equipment production, the definition of the Equipment Test Procedure and finally the equipment qualification.

Test Rig Development

In parallel to hardware and software development, the necessary test rig facilities will be developed. In order to assure a full match with those two other parallel activities, a similar approach will be adopted. It will start with the definition of the Test Rig Top Level Design Document, that will be assessed by the **Preliminary Design Review (PDR).**

After PDR authorisation the Test Rig Detailed Design Document and the Test Rig Interface Design Document will be defined concluded by the **Critical Design Review (CDR).** Final CDR authorisation will initiate test rig production or upgrade and test rig acceptance.

Software Development

Software development will cover the design, implementation and part of integration/testing of the software product reflecting completely the specified requirements. For each computer the preliminary design of the software will be defined resulting in the Software Top Level Design Document, the Software Test Plan and leading to the **Preliminary Design Review (PDR).**

After PDR authorisation the detailed design of the software will be established resulting in the Software Detailed Design Document, the Interface Design Document, the Database Design Document (if applicable), the Software Test Description and will be concluded by the **Critical Design Review (CDR).**

Final CDR aurhorisation will initiate coding and unit testing of the software resulting in the Source Listing(s) and Source/Object Code, followed by component (CSC) integration/testing of the software.

The latter will result in Source/Object Code modifications and the Software Test Procedure and will be concluded by the **Test Readiness Review (TRR).**

Equipment Testing

According to the Equipment Test Plan, the Equipment Test Description and the Equipment Test Procedure, equipment testing will start with engineering tests and lead to the qualification tests in order to obtain the preliminary clearance for flying.

If the flight test results show performance to the specification the final Declaration of Design and Performance (DDP) will be established and authorised.

Having passed all these steps the equipment will be cleared for full operational use. In case of embedded computers with loadable software a final clearance in general can only be given after hardware/software integration testing which will be carried out during system integration/testing.

Software Testing

For all computers in the system this step of testing will be carried out individually, testing the complete software product (CSCI) residing in one individual computer. Testing will be concentrated on showing that the implemented software satifies its specified functional/performance requirements, resulting in the Software Product Specification, the Version Description Document and the Software Test Reports.

System Software Testing

System software testing will be performed using all computers, all software and all interconnections of the computing system to assure that not only single computer programs but also the system software as a whole will perform as specified.

Engineering tests and then qualification tests will result in the Version Description Documents and the System Software Test Reports, and will be concluded by the ***Functional Configuration Audit / Physical Configuration Audit*** .

System Integration Testing

System integration/testing will cover HW/SW integration, subsystem and system testing, comprising the definition of the System Test Procedures, engineering tests, qualification tests, definition of the System Test Notes and definition of the Flight Release Document.

Only after completion of all of these tasks, the next phase "flight testing" will be started.

Flight Testing

The main goal of flight testing is the validation of system performance against the System Specification in the real environment. It comprises the definition of test flights, the test flights themselves and the Flight Test Reports.

A **Formal Qualification Review (FQR)** will be held after completion of the flight tests to evaluate the test results and release the system for operational service. At completion of the review, the ***Product Baseline*** is established.

Operational Service

In this phase the final product of development - as defined by the ***Product Baseline*** - is being used by the customer. This in general will result in maintenance and improvement activities. The procedure for these activities is essentially the same as above complemented by a change control procedure. This change procedure allows to identify the phase from where onwards changes in the documentation / products are required and to record these changes for all affected phases. Thus, in general not the complete system development life cycle will be run through. For example a software bug might just affect detailed design, implementation, software testing and system integration testing on ground. A new system or software baseline is then defined as the old baseline plus a controlled collection of changes.

3. Systems Engineering and Software Development

As shown in Fig. 1, we have established two handover points from systems engineering to software development and vice versa:

- after completion of the Software Specification Review (SSR) the Software Requirement Specifications together with the Equipment Specifications form the basis of software design;

- after completion of software testing the software is delivered to systems engineering for system integration testing together with all equipment.

In large avionics projects where hundreds of development engineers are involved the organisational structure has to reflect the necessity to split the development tasks between different groups and, of course, between the weapon system company and different suppliers. Especially there are systems engineering groups and separate software development groups besides others. Therefore the handovers are not just transitions from one phase to another, in addition work is transferred from one group to another. The technical knowledge and background of these two groups being different, much care has to be taken to ensure that the transitions are controlled properly. How to overcome problems resulting from the different "languages" the systems engineering people and the software developers are speaking is described in the following.

3.1 The Software Requirements Analysis to Software Design Transition

What software designers would like to get before starting the software development are Software Requirement Specifications that are complete, unambiguous, understandable and written in a formal specification language. The specifications should not change at all throughout software development. In reality development tends to be different from this ideal case for a number of reasons.

First of all the Software Requirement Specifications have to be updated during software development because e.g. the customer introduces late requirements or hardware development necessitates changes. Completeness may also be difficult to achieve because of e.g. a not yet precisely defined hardware interface. The specifications might be unambiguous to the systems engineer but not to the software designer because of his different background. On the other hand, trying to make the requirements specification too formal contradicts the way of thinking of a systems engineer. Last but not least, even if the Software Requirement Specifications would be ideal there might be a need for changing them when implementing the software. Consider the simple case that the software just gets too big to fit into the dedicated computer - a situation not unusual to avionics computers because memory is still a restriction nowadays. In order not to change the hardware one might decide to reduce the software requirements.

So, what did we do to improve this situation? In order to reduce communication problems the senior software designers were involved during the software requirements analysis phase. In fact, they wrote part of the software requirements themselves. In addition, the systems engineers participated in several design audits and especially in the Preliminary Design Review (PDR) and the Critical Design Review (CDR). Also a lot of day to day communication was stimulated between the two groups.

To prevent at least part of the implementation problems and to be more specific when writing software requirement specifications we decided to go for a specific kind of prototyping when defining the subsystem and software requirements. For that purpose the System Prototyping Rig (SPR) was built (see Fig. 2). It consists of a number of graphic workstations, microcomputers, real aircraft computers and displays as well as a fully operable cockpit. The different equipments can be connected in a very flexible way via a connection matrix and via different busses to simulate any avionics system configuration required. The software offers an aircraft model, several sensor simulations, powerfull graphics for the cockpit displays, a software development environment for all computers and a powerfull test environment.

The SPR helps in verification and experimental tests of new system architectures and in demonstrating system performance. It is also used for the evaluation of software development environments for target computers and of test support software. New equipments can be tested in a early stage in a realistic system environment. Last but not least critical software modules can be tested and their user interface can be checked by aircrews. From our point of view, system prototyping is an essential means to provide a sound empirical database for full scale development, i.e. when the system development is covered by a fixed price contract.

In addition to a complete set of documentation in accordance with DoD Std 2167 it proved to be very important to have tool support for cross referring of software requirements to design documentation. Cnly then useful verification of the design of complex avionics systems may be obtained.

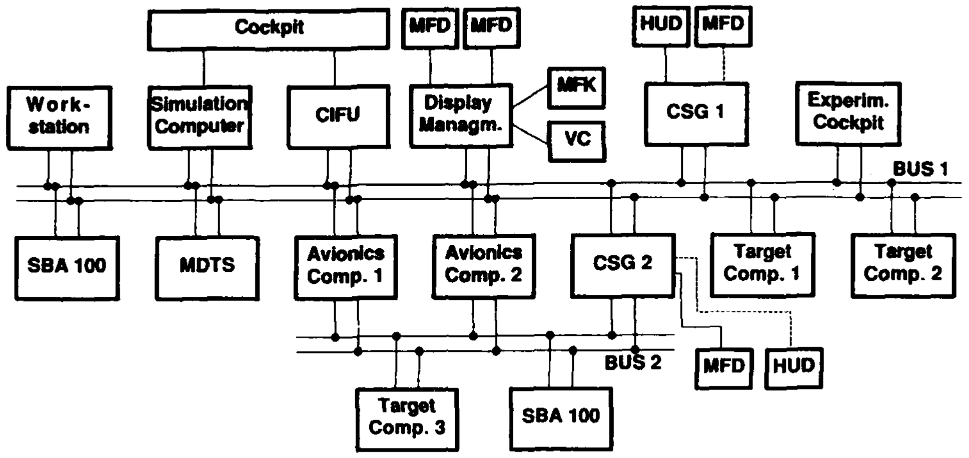


Fig. 2 System Prototyping Rig

There are other measures we adopted like rigorous configuration control or tool support wherever possible. They are nowadays widely used and we therefore do not want to expand on them here. However, it might be interesting to add a few remarks for international programs like Tornado or EFA. Part of the work is carried out in international, centralised teams; the other part is subdivided into workpackages for the participating companies. In order that this workshare be successful, an international coordination body has to be established and the interfaces have to be clearly defined. For the transition from software requirements to software design one has to be even more careful. A centralised international engineering team has to collect all the software requirements, harmonise them and define the baseline for further work. Only then each software development team for each computer is allowed to start software design. Any change of the baseline has to be carefully controlled by the centralised team and incorporated by the software development team after authorisation only.

3.2 The Software Testing to System Integration Testing Transition

State of the art avionics computing systems consist of a number of embedded computers linked via busses. When starting system integration systems engineers would therefore like to have error-free, fully tested software for each computer delivered all for integration testing at one point in time. With the classic approach - test the software for each computer on a single computer test bench, then test all software and all hardware on a system integration test bench (which we call "test rig") - responsibilities of software developpers as opposed to sytems engineers are clearly separated, but system integration in general detects a number of software problems caused by the real time behaviour of the system not anticipated in software testing.

In order to avoid these problems as far as possible we introduced an overlap of pure software testing and pure system integration testing called system software testing. This testing phase is carried out using all or at least most of the avionics computers plus some additional key hardware like displays to perform a real time system software test. Both software developers and systems engineers participate in testing to be able to separate problems caused by software from the ones caused by hardware. Thus the software is finally handed over to the system engineers only after this phase has been passed successfully.

For efficiently supporting this type of testing we built a so called Avionics System Software Rig (ASSR for the weapon system Tornado, see Fig.3). It consists of all relevant avionics computers connected as in the real system, all controls and displays essential for this testing, plus software loading and test support devices. In case of problems with a specific computer and its software the corresponding computer test bench may be linked to the ASSR to allow for indepth testing of this specific item.

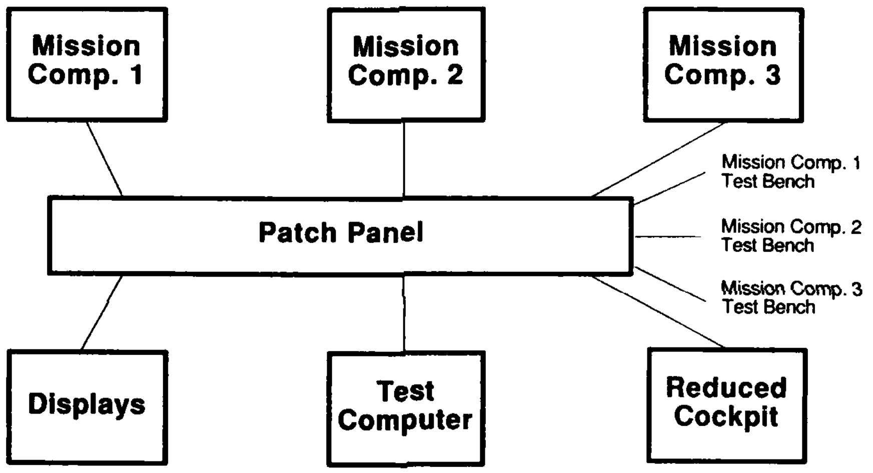


Fig. 3 Avionics System Software Rig

4. Experiences from Current Projects

The methods and tools described above have been used and are still being used in three major aircraft programs of our company:

- F-4F Improvement of Combat Efficiency; a GE only program involving the integration of a new radar, of a new A/A missile, of an additional mission computer and the improvement of the navigation system;

- HARM integration with IDS Tornado; a partly trinational program involving integration of an anti-radiation missile, of an improved missile control computer, of an improved radar warning equipment and of an improved stores management system;

- Electronic Combat Reconnaissance Tornado (GE); a partly trinational program introducing a new Tornado variant for electronic combat and penetrating reconnaissance.

Experiences gained in these programs having relevance to software development will be given below.

For systems development as a whole, a "front end investment" of effort is necessary to reduce technical risk, to deliver the required quality and to get realistic development schedules. System and software prototyping are therefore indispensable. W.r.t. software we are increasingly using the System Prototyping Rig to prototype critical software modules and to evaluate the performance of software development environments before applying them to real projects.

It is also very important in early phases of a project to define how the performance of the system will be demonstrated to the customer, because this will influence both system/software design and overall costs. As such, test tool requirements -both hardware and software- have to be defined in parall to the software requirements analysis and the Software Test Plan has to be established during the software top level design phase.

Modern software development methodologies tend to postpone real time aspects to the detailed design phase. This might be appropriate for business computing, but for avionics applications this proved to cause problems. There are cases where the software design had to be radically changed because of too extensive hierarchial decomposition and the resulting execution time overhead. In another case, the simplest and most straightforward software design led to prohibitive execution times and an increase in data records required. As mentioned before, the methods and tools do not support effective control of real time behaviour during software requirements analysis and software top level design. One therefore has to extrapolate this aspect from known systems during these early phases; or for completely new systems carry out a prototyping exercise for critical modules.

There is another potential problem when using state of the art software development tools in that they support the documentation of the lowest levels in great detail, but facilities to support an easy understandable, complete overview are missing. One of the reasons is that the tools in general store information in an object oriented way, i.e. objects being functions stored according to their place in the overall functional hierarchy. Combining the description of each of these functions into one document (for example the Top Level Design Document) -a facility the tools are offering- does not necessarily lead to a readable document. One major element of a good document is not provided automatically by the tool: a useful introduction/overview. In addition, such collections of functions tend to get prohibitively large (one of our examples: Software Requirements Specification: 400 Pages; Top Level Design Document: 2000 pages; Detailed Design Document: 6 700 pages!) and very hard to read. As a basis for the Critical Design Review or the Detailed Design Review they are of very limited use only. One therefore has to carefully control that the information of the system is stored efficiently in the tool database. A useful overview may be obtained by describing the full system in the first or first two levels of the design hierarchy down to a useful depth and thus manually introducing redundant but urgently required information into the database.

System Software Testing calls for a precise synchronisation of the software development in each computer: all software packages of all computers have to arrive at the same time. This is not realistic. The testing method applied therefore has to be flexible enough to allow for partial system integration by simulating the not available computers. The prototypes created during the initial system phases may also be helpful here.

When using the methods described in para 2 & 3 supported by appropriate tools one carefully has to consider computing power right at the beginning because computers used nowadays for avionics still have limited ressources, i.e. memory capacitiy and computing speed. On the other hand the methods/tools are demanding in these ressources. One complex task originally written in an optimised Assembler has been rewritten for a new computer in one of our projects. The first try resulted in a memory increase from 25 KW to 200 KW and an increase in execution time from 10 msec to 300 msec. Execution time overhead has been decreased considerably in the meantime, but memory overhead is essentially the same. Avionics systems with embedded computers should therefore be designed with maximum growth protential to avoid costly iterations.

The experiences gained so far indicate a significant improvement of productivity when applying methods and tools as of para 2 & 3. Explicit figures are available for the software requirements analysis phase. Comparing the costs for one page of a specification, the costs dropped by a factor of 3 to 4. On the other hand, due to generous formatting and additional information provided within the specifications, they are from 1.5 to 2 times larger than they were before. The productivity increase therefore is at least a factor of 1.5. The costs for software design, implementation and test are difficult to compare at present. But in the near future we will have a nice test case: the same requirements will be implemented in one computer in the "old fashioned" way in Assembler and in parallel in another one using the new methods and a high order language.

In order to carry out system and software development in a cost effective way, everybody involved must have a full understanding of the development process/ model and of his contribution to it. This requires besides training enough time to get acquainted to the new methods before starting time critical projects. In addition at least some of the systems engineers have to be familiar with the principles of software development and some of the software developers with the principles of systems engineering. The more information is exchanged between them the better the final product will be.

Finally let's consider some additional experiences gained in international projects. First of all, because in general part of system and software development are located at different sites (partner companies) it is essential to introduce the same methodology, the same understanding of it and the same tools at all sites. The workshare of centralised teams and the partner companies have to be precisely defined and known to everybody involved.

So far, international software for one computer has always been designed, implemented and tested in one centralised team. Applying a more rigid approach nowadays, we are trying to at least weaken this requirement. Whether this will be of success has to be demonstrated yet. On the other hand software requirements analysis and system integration test have been carried out in a decentralised manner for several years.

Defining Software Requirement Specifications at different sites asks for an efficient distributed database. A direct link of the host computers at the different sites has not been realised so far due to security problems. The way being pursued at present is to have local databases at the development sites that are integrated from time to time by a central team to form the central master database. The latter is then distributed to all sites and forms the basis for the next development step.

5. Potential Improvements

When considering the experiences we have got with our methodology and tools for system development, the following areas for potential improvement in the software development field emerge.

The methodology as such proved to be very useful and only minor corrections are deemed necessary. However, the support of the tools for this methodology is best for the coding phase and decreases steadily the more apart the considered phase is from this one. To start with real time aspects, we would like to have a better method and a tool supporting us in providing estimates for throughput and memory requirements as early as possible. In addition, the tool support to describe and analyse real time features in the software requirements analysis phase has to be improved.

The documentation the tools are providing is another area that asks for improvement. The documents should not just consist of a collection of hierarchically ordered objects and a list of contents, they should be structured in such a way, that both an easy readable overview and all the details are available. This is especially important for the Top Level Design Document in order to be able to carry out a useful Preliminary Design Review (PDR).

At present the verification of the performance of the system or the software is a very hard job, essentially carried out manually. The complexity of nowadays systems and their software asks for a considerable improvement of the tools to allow for a cross check of requirements and the corresponding test procedures, to automatically derive test data sets and test steps out of software requirements specification or the software design specifications.

Last but not least, the user interface of tools needs to be improved. A system engineer drawing data flow diagrams on a sheet of paper in 10 minutes will not be prepared to use a tool for the same purpose, if it then takes half an hour or even more. If he in addition might have to use different user interfaces for different phases of development this becomes even worse. So what is required is a universal, easy to learn user interface for all phases of development.

We hope that this contribution can help to stimulate the discussion between the systems engineer and the software developer and provides some hints for improvements.

DISCUSSION

A.Bloomfield, UK

What specific tools are used by MBB to support documentation?

Author's Reply

For Software Requirement Specifications we are using EPOS and PROMOD.

M.Stoll, Fr

You mentioned a specification written in a formal language. Can you expand on that?

Author's Reply

We used Structured Analysis as a method. The corresponding tools like PROMOD, which we are using, need a unique, unambiguous representation of the system or the software requirements. The formal elements used are data flow diagrams and data dictionaries.

EMBEDDING FORMAL METHODS IN SAFRA

Dr.Andrew Bradley
Principal Engineer
Software Technology Department
British Aerospace Military Aircraft Division
Warton Aerodrome
Preston
Lancashire
PR4 1AX
England.

SUMMARY

The SAFRA software development method has been used *extensively and successfully for the production* of real-time avionic systems at BAe Warton. The method couples a powerful, yet simple, semi-formal specification (CORE) and design (MASCOT) approach to provide an environment covering the complete lifecycle. Embedding formal methods into this established approach will combine requirements capture and structuring techniques with mathematically formal specification and designs, facilitating mathematical proof of safety critical elements.

Following an overview of SAFRA this paper provides a brief introduction to formal methods and identifies Z, a method founded on set theory and logic, for detailed investigation. An interface between semi-formal and formal techniques is defined and the results of applying the combined method to a number of avionic specification studies are summarised and discussed. The paper concludes by considering the potential benefits and costs of adopting formal methods on large scale, avionic software projects.

INTRODUCTION

The realisation, over a decade ago, that the proliferation of embedded system software must be matched by increasing productivity levels has provided continuous motivation for British Aerospace Warton to appraise, adopt and evolve effective software development techniques. The cost leverage inherent in an ability to eliminate errors early in the software lifecycle, using more formal methods of expression, led to an approach entitled Semi-Automated Functional Requirements Analysis (SAFRA Ref 1). This approach was matured and proven on a large scale project called the Experimental Aircraft Program (EAP), a collaborative European project undertaken to demonstrate the advanced technologies required for the next generation of fighter aircraft. These technologies included multi-function displays and controls in an all electronic cockpit and a Utility Systems Management (USM) system including fuel gauging, propulsion, hydraulics etc. The EAP computer hardware system architecture comprised a suite of general purpose microprocessors communicating primarily via MIL-STD-1553 serial data highways. (See Ref. 2 for a detailed description).

SAFRA is now regarded as a 1st generation approach, the benefits of which have been clearly established and quantified. Over recent years the general development of languages and environments has combined with enhancements to SAFRA to evolve a 2nd generation approach, currently being transferred onto projects such as the European Fighter Aircraft (EFA). Increasing the levels of expressed formality used in requirements analysis and specification is one area of development which has been partially implemented in the 2nd generation approach and which will fundamentally underpin advanced 3rd generation techniques such as rapid prototyping, animation and large scale re-use. (Ref. 3). Further advances in early elimination of errors, and hence productivity, will accrue from these advances.

As a driver for increasing levels of formality within requirements and specifications, the desire to improve productivity is more than matched by the obligation to eliminate software faults from systems whose functions are depended upon for safety. Software in such systems is termed 'safety critical'. Precise, mathematically formal notations are becoming widely recognised as being the most appropriate for the expression of safety requirements. They are also regarded as prerequisites for attempted proofs of software correctness.

This paper describes some salient aspects of work performed recently at British Aerospace (BAe) Warton directed towards integrating a mathematically formal notation for software specification with an established semi-formal approach. The SAFRA techniques are outlined with an explanation of the key concepts. Requirements expression aspects of SAFRA are highlighted in the form of CORE - Controlled Requirements Expression. A brief introduction to mathematically formal software specification methods is given. The relationship of a particular method, Z, to CORE is explored. A combined method is proposed - FORMAL CORE - and its application to evaluatory but representative case studies is detailed. Past experience of the introduction of new methods onto projects is extrapolated to anticipate likely costs of adoption and these are weighed qualitatively against potential benefits.

SAFRA

The philosophy adopted for SAFRA was that each phase of the software lifecycle must be supported by a method and that each method or technique is supported by a corresponding tool. The methods and tools used to develop the EAP software are shown in Figure 1. Since BAe was essentially its own customer on this project no external requirement or specifications existed. Systems Engineering departments produced functional requirements and software specifications. There was then an interface to Software Engineering departments who embodied those requirements through basic and detailed design into code. Where no methods or tools existed new ones were developed. In particular, considerable development was undertaken to produce CORE, supported by a workstation. CORE embraces the lifecycle from functional requirements through to the production of detailed designs. On the other hand, where tools were known to exist, they were imported and integrated. A brief overview of the methods and tools other than CORE is given here, the CORE method is described in the next section.

In addition to syntax and other checks which can be performed on the Workstation, CORE products were subjected to a more detailed analysis using a tool called PSL/PSA. (A product of META Systems Ltd. (d/b/a ISDOS Inc.)) PSL is a computer processable language, capable of describing a variety of behavioural and architectural aspects of systems with a range of object and inter-relationship definitions. PSA allows an overall system requirements database to be established and provides checking and analysis facilities in the form of various reports.

The software design phase invoked the continuing use of CORE notation to produce completely structured detailed specifications within the framework of a rationalised executive and high order language. The former was the Modular Approach to Software Construction, Operation and Test (MASCOT) (Ref. 4) and the latter for EAP was PASCAL.

In SAFRA the output of the requirements phase using CORE was integrated in such a way, that a basic design was derived by transforming CORE requirement thread diagrams into a MASCOT design. Experience indicates that sufficient systems analytical and structuring work must be done to ensure ease of transition at this stage. Detailed design for the individual processes (MASCOT 'activities') involved what is in normal terms 'structured program design' but in CORE terms is called layered decomposition.

The software basic design, detailed design, construction and test phases made use of the PERSPECTIVE software programming support environment (a proprietary product of Systems Designers PLC.). PERSPECTIVE is aimed specifically at supporting the development of software for embedded computer systems written in PASCAL with real-time extensions and provides a multi-user/multi-access host/target environment for large and small programming teams.

To ensure complete support for software/hardware integration and in particular the support of run-time system development, embryonic target workstations were included in the approach in the form of 'Micro-processor Development System (MDS) equipment, which provided important performance and analysis information.

CORE

The backbone of the EAP methodology and toolset was CORE. (Ref. 5). CORE is a method of expressing and analysing requirements in a controlled, structured and primarily diagrammatic manner. The method comprises 11 logical steps which when applied to a requirement will decompose it into lower level components to which, in turn, the method can be applied. The 11 steps are listed and a complete description and comparison with other techniques given in Ref. 6 - only a brief summary is given here.

The diagrammatic notation of CORE is based principally on the depiction of data flow between processes. As shown in Figure 2, processes are represented by boxes with input data lines terminating on the left edge, output data lines originating from the right edge and control (or stimulation) data lines terminating on the top edge. All data lines are classed as either critical or non-critical. Critical data flow between processes implies that sender and receiver processes act in a constrained way so that every data item 'sent' is 'received' and used. Non critical data flow places no restriction on the activation of sender and receiver processes. Whilst, at first sight, this does not appear to be a strong concept, it is a key to CORE's utility, particularly when applied to structuring the requirements of large, real-time software systems.

The CORE method, which results in a diagrammatic representation of requirements, embodies a number of key concepts - viewpoints, tabular entries, data decomposition, threads, combined threads, layering, structured design note, operational diagrams. Those concepts most relevant to the subsequent discussion on formal methods are picked out and briefly described here.

When capturing the requirements of a system two types of viewpoint are used. Firstly the direct, external influences on the system (e.g. users, other systems, ...) are grouped into 3 or 4 'bounding viewpoints'. (Figure 3). For each such viewpoint, and for the system itself (also treated as a viewpoint) a 'tabular entry' is created, comprising a list of high level actions performed by the viewpoint. Each action is annotated with high level or abstract data items consumed and produced, together with their source and destination viewpoints. In this way information is gathered about the system, its environment and information flow across the system boundary. Secondly the system itself is decomposed or partitioned by identifying 3 or 4 'defining viewpoints' within it. This process continues hierarchically, each viewpoint being split into 3 or 4 further viewpoints, so that a viewpoint tree structure is created within which requirements can be expressed and reconciled. (Figure 4). Working down the tree tabular entries are created for each viewpoint, thus gathering information about the functions of the system in a logical and controlled manner. Information flow between viewpoints at the same level is checked for consistency and, where appropriate, data decomposition diagrams are provided to display the internal structure of abstract or high level data items. Final stages of viewpoint selection may involve identification of hardware sub-systems or components.

Viewpoint selection, at any level, stimulates the creation of tabular entries - lists of actions which each viewpoint performs - at an appropriate level of detail. Subsequent steps in the method investigate relationships between and ultimately modify these actions, both within and across viewpoints.

Within a viewpoint every data item in the tabular entry is considered and categorised critical or non-critical. There may then be some actions within a viewpoint which are interconnected by critical data. Where this occurs these processes are replaced by a single action or process. The resulting set of actions are called threads - there is no critical data flow between them and some may be composites of original, tabular entry actions. This exercise, therefore, identifies parts of the system (within a viewpoint) which can operate without close synchronisation, since these parts will only be interconnected by non-critical data. The execution of the receiver process or activity is not tied to the execution of the transmitting process or activity.

When actions have been reconciled to threads within all 3 or 4 of a group of viewpoints the exercise is generalised to identify 'threads of threads' (combined threads) comprising threads within different viewpoints linked by critical data flow across viewpoint boundaries. (Figure 5). This activity pin-points areas of functionality in the system which require to be closely coupled by buffers, queues or interleaving execution, and where data are preserved as they flow from one activity to another.

Within CORE the functional behaviour of processes is specified using a combination of two techniques, graphical and algorithmic. The graphical notation extends the basic 'box and line' format already described with additional symbols and constructs encompassing box decomposition, iteration, mutual exclusion and functional equivalence. Each diagram box is also supported by a 'structured design note' wherein algorithmic relationships between process input, output and local data items is recorded. The algorithmic notation may be descriptive, narrative at high, abstract levels of the viewpoint tree or it may be a detailed, pseudo-code representation of a function at low levels of the tree. This combination of diagrammatic layering and algorithmic notations is used to specify requirements at a detailed level, within a framework of viewpoints and threads which encapsulates requirements for synchronisation, coupling and temporal interactions of the basic processes or actions.

Whilst there are a number of additional aspects of the CORE, 11 step method which cannot adequately be detailed here, the points most relevant to subsequent discussion have been highlighted. In summary CORE is a structured method within a primarily diagrammatic notation. When applied to real-time control systems and their environments it facilitates and supports the capture of functional requirements, the definition of software requirements and the decomposition of the overall software function into loosely coupled processes.

FORMAL METHODS

Steadily over a number of years, and prolifically over recent years, new methods of software specification and development based on pure mathematics have been emerging from academic and industrial institutions. The motivation for employing mathematically formal notations and methods to the specification and development of software is identical to the motivation for employing structured methods such as CORE - namely the removal of errors early in the production process both to improve the quality of delivered software and to increase the productivity of software engineers.

The term 'formal method' is widely used in varying contexts. Hence we list the principal aspects or components of a formal method as discussed in this paper.

- A mathematical notation and set of rules
- A specification structuring method
- A method of proving completeness and self consistency of a specification
- A method of 'refinement' whereby an abstract specification is transformed or reified into a programming language implementation
- A method of proof of refinement steps

The branches of mathematics which are common to most of the notations and methods developed so far are:

.	Logic and predicate calculus -	as an enhancement of Boolean operators, types and rules common to many programming languages
.	Set theory -	as a method of specifying operations on collections of data in a manner which is independent of the method of storage of the data in a computer

The key to effective use of these branches of mathematics is abstraction - i.e. the statement of a requirement in a form which is independent of the notations and restrictions of any software implementation language - the use of a 'problem notation' rather than a 'solution notation' - the oft quoted (but succinct) 'what' rather than 'how'.

In addition to improving the quality and precision of specifications, formal methods offer the prospect of combining with programming languages, proven to be mathematically self-consistent and well defined, to allow mathematical proof that a 'solution' embodied in a programme satisfies a requirement stated in a formal specification notation. Whilst currently feasible only for small systems, the prospect of proving an implementation of a safety critical function against a clear, precise and concise specification is a strong motivation for active involvement in this field. The increasing adoption of computer hardware in safety critical avionic systems has generated such involvement at BAe Warton.

It is not possible here to give a comprehensive overview of the formal methods field, rather we concentrate on what is widely regarded as the most mature class of methods - the model based specification techniques, of which the Vienna Development Method (VDM - Reference 7) and Z (Reference 8) are the most extensively used in industry.

The model based techniques have a common structure which involves the definition of a 'state' to model the internal structure (or memory) of the system. (Figure 6). The state is modelled as abstractly as possible using mathematical sets or higher level objects such as sequences and mappings. The aim is to encapsulate essential features of the system without introducing unnecessary implementation details. Operations are defined which may depend upon and affect the values of the state components and may also depend upon 'external' input variables and may generate 'external' output variables. This structure is common to all types of applications, from transaction processing and database systems to plant control and monitoring.

After detailed appraisal, Z was selected as the subject of an extensive programme of work aimed at investigating its relationship to and possible integration with the CORE based SAFRA methodology. Z is a product of the Programming Research Group at Oxford University and comprises a primarily set-theoretic notation extended to define relations, functions, sequences, etc. and associated operators. Data typing in terms of these entities is complemented by propositional and predicate calculus to yield a notation which is expressive and extendible.

Particular strengths of Z are its structuring facilities and its presentational clarity. Structuring is provided by a 'schema' notation (Figure 7) whereby a complete specification is constructed by defining and combining a number of schema operations, each representing some partial aspect of the overall requirement. A Z schema comprises two parts

. the signature - a declaration of the names and types of variables involved in the operation

. the predicate - a mathematically stated relationship between the elements of the signature

The predicate often takes the form of an explicit 'pre-condition' - a statement which must be satisfied for the operation to be invoked - and an explicit 'post-condition' - a statement describing the effect of the operation. The set of partial operations can be formally analysed for completeness and self-consistency to produce a robust set of operations embodying the requirements. Presentational clarity derives partially from the schema structuring and partially from the interspersion of descriptive, narrative text, used to complement the mathematical formality and render the specification 'readable', even by those unfamiliar with Z.

Whilst enjoying increasing use in industry the Z notation and method are not rigorously standardised nor adequately supported by tools. There are also strict limits to the scope of applicability of the method, especially in the area of real-time control systems. Large scale structuring, process synchronisations and temporal aspects in general are not supported.

In summary formal methods offer a further increase in expressional formality beyond that already achieved by the adoption of structured methods such as CORE. The model based or 'constructive' class of methods is the most established and widely used. Z was selected by BAe Warton for detailed evaluation and possible integration with CORE.

FORMAL CORE

It is clear from the previous two sections that the strengths of CORE may be complemented by those of Formal Methods in the application domain of large, real-time avionic systems.

CORE is a powerful, structured method for the controlled and modular development of system requirements and specifications with particular emphasis on module interaction synchronisations and temporal relationships. Mathematical formalisation of the temporal interactions of processes is still a research area, however the loosely coupled processes defined by CORE analysis are potentially amenable to established formal specification methods. On EAP the functional content of CORE threads was specified informally (descriptive narrative) high in the viewpoint hierarchy (i.e. early in the lifecycle) and semi-formally (diagrammatic decomposition and pseudo-code) later in the lifecycle. The use of Z allows precise, formal definition early in the lifecycle followed by potentially verifiable refinement thereafter.

In this way the concept of FORMAL CORE was evolved as an enhancement to SAFRA which combines the strengths of structured and formal approaches. An interface is proposed at the thread level so that all temporal aspects of requirements are captured within CORE and all functional aspects of threads are specified in Z. In order to investigate this concept a number of representative threads were selected from the EAP Utilities Systems and formally re-specified in Z, based on an understanding of the engineering requirements (gained from existing specification documentation and interviewing engineers). The specifications were then refined to code (PASCAL) before a detailed comparison of both specification and code was made with the original, CORE-based, EAP documentation.

Although the Utilities Systems threads chosen for study were quite small (<100 lines code when implemented) clear indications of the benefits of formality were seen. These prompted further study in the form of a relatively large (2000 lines of code) case study which had proved particularly difficult to specify and design using standard CORE - namely Soft-Key decoding for the multi-function displays in the all electronic cockpit of EAP. Combining the results and conclusions from this series of initial case studies the following main points emerged:

- the strengths of the formal notation and method (precision, clarity, verifiability and specification by construction) complement the strengths of the CORE semi-formal notation and method (organised requirements capture, techniques for handling dynamics, temporal aspects, concurrency)

- embedding formal methods in an established, structured approach is a viable way to feed formality from a research and development environment through to projects

- the formal definitions of a domain vocabulary - i.e. a set of formally defined functions which are specific to the application and used frequently - helps to conceal unnecessary detail in specifications and provides a natural working language for systems engineers.

- in all the case studies the use of the formal method resulted in enhanced clarity, visibility and traceability (therefore vastly improving the prospects of effective maintenance). Also, notably these benefits were not at the expense of code size, performance, ... but in fact were accompanied by a reduction in code size together with reduced stack usage.

- more effort is expended on requirements capture/specification when a formal notation and method is employed. Less effort is expended on design and coding - the refinement from specification to code being relatively straightforward for avionic systems which are not dominated by complex data structures.

- there is a marked lack of mature tool support for creating, analysing and refining formal specifications.

- there is a barrier, at best only perceived but probably real, on the part of software (and more so systems) engineers to the introduction of mathematical techniques into the software production process.

In addition it was established that the formal notation enables extensive formal analysis of safety critical system specifications to be performed and that formal specifications are pre-requisites for the application of automated code analysis tools. Since a total verification scheme for safety critical avionic software is a prime goal of research at BAe Warton we expand here on the verification aspects of the case study findings.

Verification through the standard 'waterfall' lifecycle is taken to be i) the confirmation that any level satisfies its specification (i.e. the next higher level) and ii) the confirmation of logical completeness and self-consistency at the highest level of requirements/specification. These two aspects are treated separately.

i) Throughout the case studies the translation from a formally specified EAP CORE thread to a functional implementation in PASCAL was found to be straight forward enough to be a 1 or 2 stage process, the two main aspects of which are logic processing and the reusable functions of the 'domain vocabulary'. Once coding structures are established for basic logical constructs (=> implies, <=> if any only if, etc..) the correspondence between logic processing specification and code can be checked rigorously by inspection. The correspondence between specification and code for reusable functions is generally more complex, involving code loops etc., and proofs involving recursion/induction are formulated. This area of work is continuing and being combined with the use of commercially available semantic analysis tools which are used to automate and assist the process of proof of code against specification.

ii) The proof of logical completeness and self consistency of a formally specified CORE thread has been addressed with encouraging results. Considerable analytical effort and expertise is required and it is considered practically feasible to apply such a depth of analysis only to safety critical functions. Nevertheless significant progress towards establishing a total verification scheme has been achieved through the investigation of formal specifications and refinement methods in conjunction with CORE.

CONCLUSIONS

Over recent years British Aerospace Warton have achieved a firm base of experience in the application of structured software engineering techniques to the production of embedded avionic software. The adoption of these techniques significantly increased productivity, enabling timely completion of the large scale EAP project. Increased productivity was accompanied by increased quality, the error rate of code delivered to test rigs being reduced by a significant factor over previous projects. SAFRA, and the more formal method of expression provided by CORE in particular, enabled the elimination of errors early in the software lifecycle and is primarily responsible for the quantified achievements.

The volume software content of current and future projects rises steeply over that in EAP and hence improved techniques must provide further improvements in productivity. The investment of safety critical functions in software is also unavoidable and hence the best software engineering practices must be adopted in order to minimise the occurrence of errors. These two requirements have motivated an active research programme to further increase levels of formality in SAFRA by incorporating an established, mathematically formal specification method, Z, into CORE.

Realistic, avionic case studies from EAP have been completed with positive indications of potential benefits. Clarity, visibility and precision of specifications was enhanced - attributes which engender the early identification of errors and inconsistencies. Traceability from requirements through to code was enhanced, thereby improving the prospects of effective maintenance. In addition a formal statement of requirements enables mathematical proof of code against specification - a prime objective for safety critical functions.

The potential benefits, therefore, are clear. The potential costs, however, are more esoteric. The introduction, by education, training and experience, of formal mathematical techniques into the software production process will require sound management and motivation. The barriers that exist have been partially overcome by the successful adoption of CORE, but the new areas of mathematics are unfamiliar and can appear hostile without careful selection of notation. In addition the further concentration of effort and analysis on the early stages of the lifecycle will make it more difficult to identify staged development milestones in a large project.

It was considered essential to progress the formal methods work by attempting to quantify the relative weights of these potential costs and benefits under project conditions. A pilot project has been identified and will be undertaken over the second half of 1988 at BAe Warton. This will include the education and training of software and systems engineers prior to their involvement in a heavily 'instrumented' but otherwise realistic project. The project will rely on a well defined FORMAL CORE method, in-house and external training courses and Z editing, printing and database tools integrated with the CORE workstation. This project will establish, quantitatively where possible, indicative costs and benefits of using formal methods in a realistic, avionic project environment.

REFERENCES

1) C. P. Price and D. Y. Forsyth, British Aerospace, 'Practical considerations in the introduction of requirements analysis techniques', 1982, 'Software for Avionics - Advisory Group on Aeronautical Research and Development - Proceedings 330'.

2) P. Cronshaw, British Aerospace, 'The Experimental Aircraft Programme software toolset', 1986, Software Engineering Journal - November.

3) A. O. Ward, British Aerospace, 'Three generations of software engineering for airborne systems', 1988, Paper 21 of these proceedings.

4) 'The Official Handbook of Mascot, Version 3.1, Issue 1' - Published by the Joint IECCA and MUF Committee (JIMCOM), Royal Signals and Radar Establishment, Malvern, England.

5) G.P. Mullery, 'CORE - a method for controlled requirements specification', Proceedings of 4th International Conference on Software Engineering, IEEE Computer Society Press.

6) M. Looney, 'CORE-STARTS Debrief Report', 1986, Published by National Computing Centre, Manchester, England.

7) C. B. Jones, 'Systematic Software Development using VDM', Prentice Hall International Series in Computer Science, 1986.

8) I. Hayes, 'Specification Case Studies', Prentice Hall International Series in Computer Science, 1987.

ACKNOWLEDGEMENTS

I would like to thank the Directors of British Aerospace, Military Aircraft Division for their permission to publish the paper.

I would also like to acknowledge the support of the Procurement Executive of the Ministry of Defence.

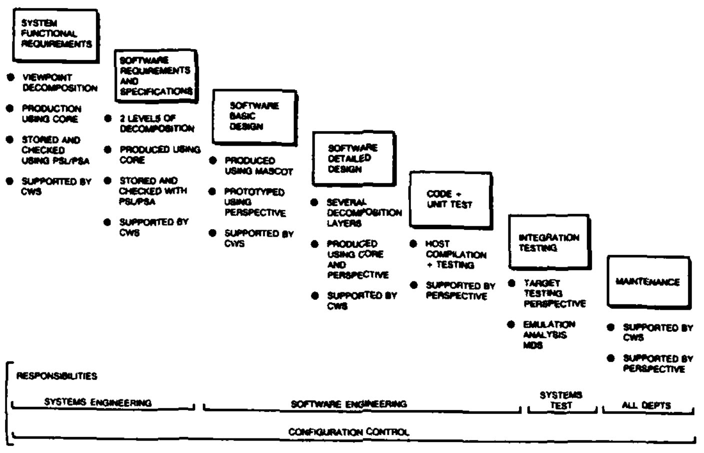

SYSTEM FUNCTIONAL REQUIREMENTS
● VIEWPOINT DECOMPOSITION
● PRODUCTION USING CORE
● STORED AND CHECKED USING PSL/PSA
● SUPPORTED BY CWS
SOFTWARE REQUIREMENTS AND SPECIFICATIONS
● 2 LEVELS OF DECOMPOSITION
● PRODUCED USING CORE
● STORED AND CHECKED WITH PSL/PSA
● SUPPORTED BY CWS
SOFTWARE BASIC DESIGN
● PRODUCED USING MASCOT
● PROTOTYPED USING PERSPECTIVE
● SUPPORTED BY CWS
SOFTWARE DETAILED DESIGN
● SEVERAL DECOMPOSITION LAYERS
● PRODUCED USING CORE AND PERSPECTIVE
● SUPPORTED BY CWS
CODE + UNIT TEST
● HOST COMPILATION + TESTING
● SUPPORTED BY PERSPECTIVE
INTEGRATION TESTING
● TARGET TESTING PERSPECTIVE
● EMULATION ANALYSIS MDS
MAINTENANCE
● SUPPORTED BY CWS
● SUPPORTED BY PERSPECTIVE
RESPONSIBILITIES
SYSTEMS ENGINEERING
SOFTWARE ENGINEERING
SYSTEMS TEST
ALL DEPTS
CONFIGURATION CONTROL

Figure 1 EAP SAFRA Lifecycle

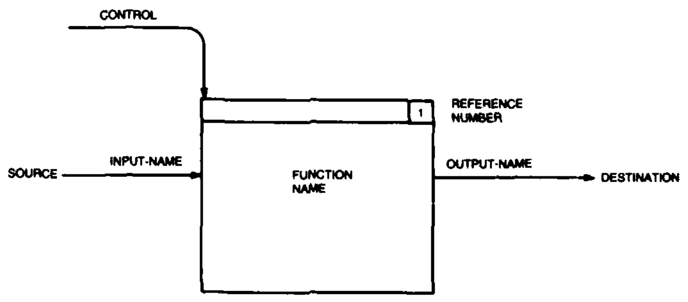


Figure 2 Basic Core Notation

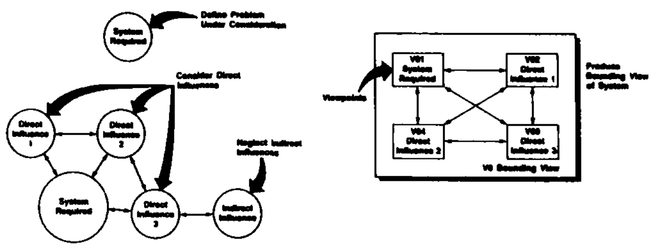


Figure 3 Bounding Viewpoints in Core

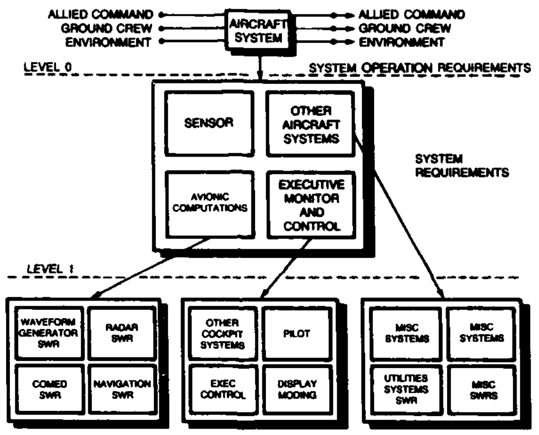


Figure 4 Defining Viewpoints in Core

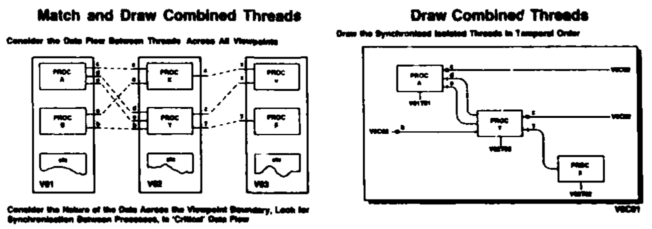

Match and Draw Combined Threads
Consider the Data Flow Between Threads Across All Viewpoints
PROC A
PROC B
PROC X
PROC Y
V01
V02
V03
Consider the Nature of the Data Across the Viewpoint Boundary, Look for Synchronisation Between Processes, ie 'Critical' Data Flow
Draw Combined Threads
Draw the Synchronised Isolated Threads in Temporal Order
PROC A
PROC Y
VSC01

Figure 5 Combined Threads in Core

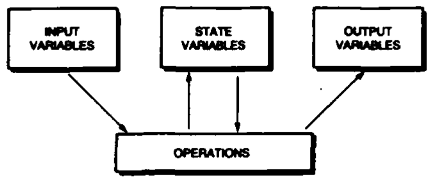


Figure 6 Basic Format of Model Based Formal Specification Methods

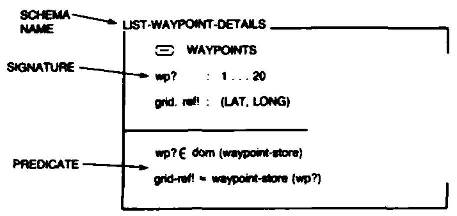


Figure 7 A Z Schema

DISCUSSION

K.Benner, US

Is there automated support for establishing and maintaining "threads" between Requirements and Specifications, in particular, one-to-many relations?

Author's Reply

In the core viewpoint hierarchy, the threads of a "requirements" viewpoint must be fulfilled by the combined threads of the constituent "specification" viewpoints — the relationship is one-to-many. The core automated support ensures that the *interface* of "requirements" viewpoint threads to their environment (i.e., other viewpoints at the requirements level) and is maintained at the specification level.

W.Mansel, Ge

How does Z relate to using a formal method of mapping a core requirement specification into a basic design in MASCOT?

Author's Reply

The strong mapping between core requirements/specifications and MASCOT design entities was developed and exploited successfully on the EAP project. The proposed use of Z with core does not affect (and is complementary to) this mapping. Z is used for the function specification of threads within core.

M.Stoll, Fr

Are you using or developing any tools that map formal language into natural language?

Author's Reply

As part of our integration of Z into core for use by systems engineers, we have attempted to formally define "language" elements which are natural to the domain of avionics control systems. The systems engineer uses these pre-defined elements in constructing a specification. No work has been performed at BAe on *automatically* translating formal specifications into natural language.

FORMALIZING A THEORY OF REAL-TIME SOFTWARE SPECIFICATION

by

Robert D. Busser
Project Engineer
Allied/Signal Aerospace
Bendix/King Air Transport Avionics Division
2100 NW 62nd Street
Fort Lauderdale, FL 33310
USA

SUMMARY

The approaches taken to the specification for requirements and design of real-time embedded systems software range from "not at all" to informal textual specifications to semi-formal graphics-based languages for data flow, control flow, state transition, data structure, etc., diagrams to formal approaches such as VDM and HOS. Each approach has some problems when applied to the specification of large software systems. Part of the reason is that there are few formally defined models of the specification hierarchy commonly associated with requirements, design, and implementation specifications. Such a model is required as a point of reference in discussions about such specification issues and completeness and consistency analysis performed by some of the current generation CASE tools. A model can provide guidance in the identification of just what information to specify; regardless of the syntactic aspects of the specification approach used to capture this semantic information. In addition, such a model can be applied as a the deductive inference model underlying an AI-assisted CASE environment capable of exercising a greater "experts" understanding of the information being specified that current generation CASE tools.

This paper presents such a model in the form of a theory of Real-time Embedded System Specification, RESS theory. A partial vocabulary of theory is presented along with the basic axioms and representative theorems.

Introduction

The intent of this paper is to present an introduction to a theory which will hopefully contribute to the understanding of the various aspects and levels involved with the specification of software based systems. In addition, a framework for the definition of consistency, completeness, and correctness for a given set of software specifications is developed.

The number of different classes of information required to completely describe a given software system leads to a theory both broad in scope and deep in complexity. A theory of specification must place all of these classes in a general framework and provide a meaningful interpretation of these classes and their interrelationships. To facilitate the application of such a theory, the theory must be described using a language as close to the language of the application domain as possible while still retaining the formal semantics required for precise description. This paper discusses the need for such a theory, presents some of the theory's basic definitions, and provides an overview of its complete scope. It also describes a potential application of the theory and discusses how the theory may be useful in day-to-day software development for real-time embedded systems.

A second title to this paper could have been, "The Tao of Software Specifications". The Theory of Real-Time Embedded System Specifications, RESS theory, is based on the interplay between two opposing, in a yin-yang like fashion, concepts; that of the syntactic and the semantic nature of information manipulation. When identifying and specifying a requirement or design, the syntax, ie., the representational and structural constraints governing associated data and transformation objects, as well as the semantics, ie., the information content of these data objects and transformations, are equally important. This duality provides a unique and analyzable point of view of the problem and permits a useful separation of specification concerns. In turn, this leads to a view of consistency and completeness which is more robust than that which is generally applied to the analysis of real-time specifications. By realizing and properly focusing on the natural interplay between these two aspects, a view of specifications can be developed which is, in a Taoist sense, in balance and harmony.

This paper does not propose any specific language or format for the representation of software specifications. There are already many such formats to choose from. Such standards as the DOD's 2167, the FAA's DO-178A, Boeing's D6-35071, languages like PSL/PSA and AXES of the HOS methodology, and methodologies as Structured Analysis - Structured Design all propose some means of requirements and/or design representation. Some of these approaches; PSL/PSA, Structured Analysis - Structured Design; are primarily format standards. In other words,

> if one has a given set of information to capture in a specification then it should be specified in such and such manner.

Others are information content standards; DoD's 2167, Boeing's D6-35071;

> one should specify the following classes of information in some acceptabl hierarchically-organized manner once they have been identified and collected.

Some of the current approaches to software specification base their representation language and/or formats on a formal semantic definitions. For example, HOS bases its requirement specification syntax on a set of algebraically defined constructs, a language called AXES, which may be used to decompose high level functions into lower level functions in a correctness preserving fashion. The lower level functions are either additional sub-functions, also specified using AXES constructs, or are primitives with obvious or axiomatically characterizable properties [1]. The decomposition process continues until a level of abstraction is reached where the correctness of the execution of sequences of operations appears to be obvious. Thus, the semantic properties of a given program specified using the approach of HOS are defined "operationally" [2]. One drawback to this approach is the difficulty in querying the specification on a detailed requirement. For example, if one wants to determine from the specification what the program is required to generate for :

> output Z in sub-mode "a" of mode "A" when the input X is 0

One must, in essence, "symbolically" execute the specification until the above situation is uncovered and the answer is apparent. It is not easy to go and look up Z and find out its relation to input X in mode A.

Other approaches to requirement specification follow a more traditional intuitive and pragmatic paradigm proposing standard templates for expressing common specification classes. These are usually based on experience developing software according to the "just start coding it up and then debug it into correctness" [3] *school of* thought. PSL/PSA proposes the specification of a system in terms of the Entity-Attribute-Relationship model of the desired system. The Problem Statement Language is a restricted set of relationships useful in describing desired relationships between input, process, and output objects of the system being specified. The Problem Statement Analyzer acts like a relational database manager over the possible relations described via PSL providing a query mechanism for retrieval of the information specified in this manner. PSL/PSA is an example of a methodology independent facility, ie. there is no formal definition of consistency or completeness properties of the resultant specification. [4].

Still other more formally based approaches have tended to leave "requirement" specification to representation in informal textual prose, if at all. They proceed to axiomatically characterize the desired semantic properties of specific programming language "implementation" expressions. Such specifications often take the form of Hoare-logics

$$\text{ie.} \quad P \; \{ Q \} \; R$$

expressions. In this case P is a precondition assertion describing the condition of the input state space which must hold before execution of the process Q in order for the execution of Q to guarantee that upon termination the postcondition R will hold over the output state space. An examples of such an approach can be found in [5]. The methodology described was derived from a methodology, WELLMADE [6], developed in an experiment in formal software development methodologies at Honeywell. This method proposed the specification of a system's design using <P, R> tuples describing the desired properties of a hierarchically organized abstract machines. Dijkstra's constructive approach, ie. the application of his "wp" predicate transformer in the derivation of a program's sequential operator sequences from the program's <P, R> tuple, was applied to assist in the determination of a "provably correct" next lower level of abstract machines.

Tool vendors supplying Computer-Aided Software Engineering (CASE) products often sit between the mathematically formal and the pragmatically informal. Most CASE tools support specification techniques which have, or could have, a formal semantics. However, the vendors often advertise that their tool provides "methodology independent" representational techniques. They suggest that users supply their own semantic interpretations of the tools graphics and/or languages. The tools seldom enforce any particular semantics within the CASE tool other than the "syntactical" semantics of the representational techniques being supplied. An example would be the level balancing of dataflow diagramming tools.

From a business perspective this may be necessary. Software developers on the whole are very particular in the area of methodology preferences. A tool which rigorously enforces a specific semantics, and thus, methodology, often has trouble selling to the general development community.

Why a Formal Theory of Specification ?

One of the prime reasons for mathematical formality in specifications is that human languages are notoriously full of ambiguities, double meanings, context-dependencies, and other such characteristics. The written forms of such languages do not have complete, and commonly agreed upon, syntactic and semantic characterizations. Such characterizations enable each expression of the formally defined language to carry exactly and only the information required for correct interpretation.

The strength of human languages is that they can convey a great deal of information with very little actual expression by counting on the receiver of the expression to complete the information transfer from his or her experience and/or context. The weakness of human languages, at least for the conveyance of precise information, is that each individual's experience or context differs; often in very subtle ways. The requirement expressions :

"The value of the output shall be computed sufficiently often enough to prevent undue ..."

"The user interface shall be easy to learn..."

"Response to changes in input value shall be fast enough to minimize the lag time through the system."

which are of little worth to the designer of a software system, are also 3 of the most often written software requirement expression schemas.

The reader of a traditional textual requirements specification brings a different interpretation context to the specification than does the writer. This invariably leads to interpretations of the specification which differ by some unknown degree. This results in a potential for errors during design which are very subtle and difficult to detect; both the writer and reader think they are in agreement, not realizing they are interpreting the same sentence differently.

Currently popular CASE tools can compound the problem of detection of differences. The major focus of the graphics-oriented tools and methods has been the syntactic aspects of their associated languages.

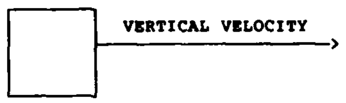
- Their syntaxes are normally well defined.
- Consistency and completeness of such specifications are defined in syntactic terms like level balancing and parse-ability.
- Syntax-directed tools support the representation and analysis of requirements specified by applying them.

This focus often results in syntactically complete and consistent specifications.

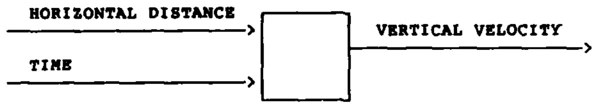
However, many proponents of such approaches associate semantic correctness with such attributes. This is generally an invalid inference. The information content of syntactically well-formed specifications may not be semantically complete and consistent with respect to the problem itself, or even other parts of the same specification. This results from the lack of a well-defined and well-supported axiomatic, denotational, or even operational, semantics for the languages on which the graphic approaches are based.

Engineers communicating via textual specifications are usually familiar with the problems of interpretation and are thus, generally on guard for such occurrences. However, when communicating via real-time-extended dataflow diagrams [7], for example, they often believe that since the graphic languages have relatively formal syntactic definitions they also have a commonly agreed upon formal semantics. Thus, they are not on guard and often miss occasions of differences in semantic interpretation.

In addition, most CASE tools providing such graphic representation techniques are said to be capable of automatically checking specifications for consistency, completeness, and correctness. For commercially available structured analysis and structured design tools, such checking is invariably syntactic in nature. They do not have the capacity for representing the semantic properties of the specification, let alone analyzing the semantics. For example, they can detect an error in a specification calling for the generation of a data object "out of thin air", ie., from no specified set of inputs.



But they can not detect an error in a specification of the generation of a data object required to represent "vertical velocity" from data objects representing "horizontal distance" and "time".



One of the reasons for this deficiency is a lack of support for expression of data type, ie., the full set of syntactic properties of a give data object plus the range, dimensionality, and possible manipulations of the values which may be bound to the object. Most only permit a rudimentary form of data object structuring in the form of traditional data dictionary entries.

An automated syntactic analysis capability, when coupled with the belief that the techniques do have a commonly agreed upon semantics among engineers following a common methodology, often leads users into a very false sense of security. This allows problems to propagate all the way to design or implementation before they are detected and dealt with.

Formal specification approaches suffer from a different class of problems. They do not suffer from problems in interpretability so much as manageability. Once an engineer learns the formal language of discourse, for example, first order predicate calculus, such specifications are no more difficult to interpret than any other formal language, such as a high level programming language. Instead, an often expressed opinion is that the drawbacks to the application of most formal methods are related to the fact the method was originally created for "toy" academic problems. Such methods are often difficult to scale upward to handle large problems.

The semantics of a single source code procedure may be specified in the pre- and postconditional assertions of axiomatic approaches to specification [8]. This, in my opinion, is not all that difficult. But the mental and physical manipulation of all the terms in the expression of the postcondition for an entire embedded system, without support of automated tools, is considered by even the most diligent and mathematically inclined to be extremely difficult and tedious, at best.

A postcondition, the logical assertion describing a relationship to be maintained between a system's input and output state space, will contain, as a bare minimum, as many AND'ed terms as there are system output objects.

In practice, a system describable in such a minimum fashion would only be capable of minimal functionality. Such a postcondition would also require pairing with a very strong precondition. Such a precondition would necessarily express the requirement that the input state space be guaranteed to never exist in a state indicating missing or corrupted data. In other words, an input state space in which all input objects are bound to values directly usable in the semantic transformations which satisfy the single term postcondition of the associated output object. For embedded systems, such a guarantee is generally impossible.

Thus, much of the software for an embedded system is geared toward adding exception handling capability minimizing the strength of the precondition such that for any point in the input state space a valid output response is given. This strengthens the ability of the embedded system to deal with loss or corruption of its input data and minimizes its dependents on its parent system.

In addition, most typical software-based embedded systems are software-based because of a requirement to support multiple modes of operation. Also because of the ease in which software can be designed to manage such changes in operational state. In such an embedded system, some number of the output objects can be expected to have different terms relating to its semantics in the postcondition for each mode, as well as different exception handling terms. Thus, the number of terms in the postcondition of even a simple embedded system is probably at least an order of magnitude greater than the number of output objects. Also, this is only at the most abstract level of specification. As the software design is laid out, decomposing the problem into lower level subsystems, the number of postconditional terms expands exponentially.

A methodology for partitioning, manipulating, and managing the massive amount formal expression must be developed. Without such a methodology, the formal development of meaningful embedded systems becomes intractable and essentially impossible. Another drawback to formal specification is that it calls for greater precision of information than informal specifications.

A methodology for formal specification must also assists in the initial identification of just what information to specify in order to support such precision. Precisely what set of information completely captures the requirements of a given problem, and what properties must be true for elements of this set to be considered semantically consistent, are questions not well addressed by the currently supported languages, tools, and methodologies.

There is also little academic work on these aspects of real-time software requirements specification to turn to. Most of the work considering semantic, as well as syntactic, properties do so only at the program source code level. [9, 10, 11]. Other work in theories of specification ignore the application world's perspective on the problem staying very academic and closely related to academically-nice, but fictional, programming languages like Dijkstra's guarded commands. [12]. No work that I am aware of discusses the special attributes and constraints of formal specifications for real-time embedded systems. Nor is there any work formally treating the real-time embedded system concepts of program design. When formal specifications approaches are discussed at all it is with respect to a "logically correct" program rather than an "efficient", "modifiable", "understandable", etc. program.

In order to define such a methodology, one must have a theory describing the objects to be manipulated by the methodology and the relationships between those objects which provides the basis of the legal manipulations which may be carried out within the methodology. The purpose of RESS theory is to provide a model from which a formal development methodology may be developed. It also provides a basis for the mathematical proof of the specification properties; completeness, consistency, and correctness; for all and/or between all abstract specification levels.

In general, a theory consists of a language and a set of axioms or axiom schemas. The language used in such a description is the language of predicate logic where the set of constant, function, and predicate symbols are restricted to a specific vocabulary, ie., a particular subset of the symbols allowed in general predicate logic. Each of the subsets in the vocabulary may be finite or infinite, empty or non-empty. The terms, sentences, and expressions of a theory are those terms, sentences, and expressions of predicate logic whose constant, function, and predicate symbols are in the vocabulary of the theory. [13].

This paper is not intended as a complete and detailed mathematical treatment of RESS theory. Instead, it presents an interrelated set of ideas which can offer many practical software specification guidelines while providing an overview of its scope and purpose.

The presentation of RESS theory follows in spirit, if not complete mathematical rigor, the general axiomatic approach to theory presentation by introducing a subset of its terms, definitions, and logical operators. This semi-formal approach will facilitate the presentation of the theory and the expression of axioms and theorem characterizing attributes of the theory and its application while permitting enough abstraction of in the language of presentation to describe the theory in terms of real-world concepts like "data object", "Function", "Architectural Design", etc.

In essence, RESS theory defines a set of valid relationships among

1. an abstraction level of requirement specifications,

2. a valid satisfaction of functional requirements by a functional design specified in terms of sequences of canonical data transformations,

3. a "fuzzy"-satisfaction of Non-Functional requirements (memory constraints, processor throughput limitations, maintainability, hardware organization, etc.) in terms of architectural design specifications that generalize and subsume the canonical data transformation sequences,

4. and the requirement specifications for the next lower level of abstraction.

In sequence, the rest of this paper introduces definitions for some of the terms and logical operators comprising the vocabulary of RESS theory. It then presents an informal expression of some of the primary axioms and theorems of

RESS theory. Finally, it discusses the application of RESS theory as a basis for an AI-based software specification assistant.

Vocabulary of RESS Theory

RESS theory describes the contents of and relationships among elements in the set of software specification classes. In order to state the axioms, theorems, and rules of RESS theory, these elements must be given accurate and consistent definitions. It is not intended that the definitions given are the only true definitions for such terms; only that they are consistent ones.

Definition : Data Object D1

An information carrying element having a specific syntactic representation, or data structure, and a specific application domain semantics, ie., its value *binding*, potential *range* of value bindings, and dimensionality. For this paper data objects will either have meaningful names or be represented by lower case letters u,v,w,x,y, and z.

In order to refer to the syntactic and semantic attributes of a given specification element or class, RESS theory defines two unary operators, sr() and sv(), which reference the syntactic representation and semantic value aspects of their single parameters, respectively.

Definition : Syntactic Representation Operator D2

The Syntactic Representation Operator, sr(), is a unary operator that returns the syntactic structure of its operand.

For example :

sr(velocity_object) = "REAL +/- 10^{-6}"

Or

sr(set_of_velocity_objects) = "Array [0..25] of REAL +/- 10^{-6}"

When the operand is a data object, the sr() operator may be looked at as returning all the information normally associated with the concept "data type" without reference to its valid range of values, its access methods, or any dimensionality. The operand of an sr() need not be a data object but can be any object with a syntactic attribute.

Definition : Semantic Value Operator D3

A Semantic Value Operator, sv(), is a unary operator which returns, or represents, the semantic value of its operand.

For example :

sv(velocity_object) = "100 kph ε {0 kph ... 500 kph}"

A very important point to notice is that the units "kph" and the potential range of values "{0 kph ... 500 kph}" is as much a part of the semantic value of the object as the "100". Also, the operand of an sv() need not be a data object but may be any object that has an intended meaning associated with it not expressed by its structural representation.

Groups of data objects present at the input and output boundaries of a system; software, hardware, or abstract; and their interrelationships form the domain of discourse for Requirement Specification for that system. It is a well accepted view that requirements express "what" a system is supposed to do to the data objects at its input boundary in order to generate those data objects expected at its output boundary. However, very often requirements are specified using an operational language which makes statements about the process, the "how", of performing the required transformations. For example:

```
Let x,y be input objects
    z   be a output object


 IF the inputs x and y have had 50 ms to settle
THEN z should be set to x + y
ELSE z should be reset to 0
```

RESS theory defines a requirement to be a relationship between a data object required to exist at the system's output boundary at a specific point in time and some subset of the data objects at the system's input boundary from which the output object is derivable.

$$(z = x + y) \ @\text{Time} \ ((\tau = \tau_0 + 50 \text{ ms}) \ \wedge \ (y \neq 0))$$
$$\vee$$
$$z = 0$$

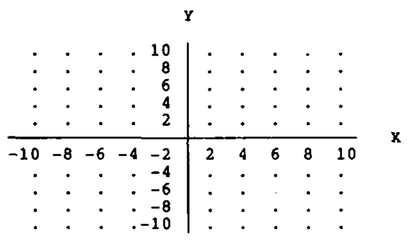
In the first approach the requirement expression implies a sequence of processing that is an unnecessary over-specification. This often prejudices the designer. In addition, the process-oriented view of the first example does not promote the mathematical reality, not just the implementation reality, that z cannot be set to x + y if y = 0. Also, the first example leads one to expect that z must be set to 0 on every examination prior to 50 ms. This need not be the case if z can be shown to have been initialized to 0 at τ_0 and is not affected by any other requirement.

In order to facilitate the description of such relationships, RESS theory incorporates the concept of "state space".

Definition : State Space D4

A Cartesian space whose dimensions (not to be confused with object value dimensionality, .ie, "kilometers per hour") are defined by a set of data objects. The scale for each dimension is defined by the valid range and precision of the value of each data object.

Thus a state space consisting of two data objects x,y of type integer with range -10 to 10 with precision of 2 would define a state space like the following :



If x and y are the only inputs to a system then each point of the above grid represents a possible state of the input state space and which the system is required to deal with appropriately.

For this paper, an input state space, consisting of all the input data objects of a given system and the vector describing the set of values which make up the current state represented by a specific point in the state space, is referred to by "I". A system output state space and vector is referred to by "O". When describing relationships involving only a single output object, which may consist of more primitive objects but is identifiable in its own right, the designation O_i is used with the understanding that

$$\forall i : O_i \subset O.$$

I_i refers to the set of all partial input state spaces on which O_i depends. When referring to the j'th partial input state space on which O_i depends in a given circumstance, I_{ij} is used with

$$\forall i,j : I_{ij} \subset I_i \subset I.$$

In addition, for any state space X :

sr(X) = the Cartesian space defined by all data objects in X multiplied by their ranges.

sv(X) = a specific point in sr(X) identified by the vector made up from the current values of all the data objects making up X.

The concept of state space provides a means by which a definition of "abstraction level" may be given.

Definition : Specification Abstraction Level D5

RESS theory defines Specification Abstraction Level to be the view of a software system through the information present in the syntactic structure and semantic value associated with a given input/output state space tuple <I,O>.

Within RESS theory, any and all information, including requirement specifications and design specifications, that involve relations among those data objects making up a given <I,O>, is defined as pertaining to the same Specification Abstraction Level. This view of abstraction is a projection on the hierarchical structure of a software system in general. It is not the same definition of abstraction associated with "data abstraction" and "abstract data types".

Many of the relations defined in this paper reference a state of a given state space as a parameter. In order to facilitate the application of these relations in examples where the data objects have been individually enumerated, a state constructor is all be defined.

Definition : State Space Constructor D6

A State Space Constructor is an operator that takes a list of individual data objects and denotes the current state and state space which results from their valid ranges taken together.

Let $y_1, y_2, \dots y_n$ be a list of data objects

Then $\Sigma_{i=1 \dots n}\, y_i$ denotes their cartesian state space

and $\Sigma_{i=1 \dots n}\, y_i = I$

$<->$

$sr(\Sigma_{i=1 \dots n}\, y_i) = sr(I)$

$\wedge$

$sv(\Sigma_{i=1 \dots n}\, y_i) = sv(I)$

In RESS theory, the definition of the term "function" is closer to its mathematical form rather than the software-traditional definition of "a process to be performed".

Definition : Function D7

A Function, $f()$, is mapping relationship between a point in the function's domain, the function's input state space vector I, and some point in its codomain, its output state space vector O.

$O = f(I)$

or

$<sr(O), sv(O)> = f(\ <sr(I), sv(I)>\ )$

RESS theory makes use of the mathematical definition of function in order to define the requirement specification concept of Functional Assertion.

Definition : Functional Assertion D8

A Functional Assertion, $O_i = f_{ij}(I_{ij})$, is the statement that at some point, or points, in time there is required to exist a mapping between a valid point in an output state space, established by the i'th output data object of a system, and a valid point in the state space established by the j'th set of input objects from which O_i is derivable. This mapping is dictated by the syntactic representation and semantic value of the object O_i, the objects making up I_{ij}, and of f_{ij}. As a matter of convenience, the following syntactic conventions is used :

$sr(O_i = f_{ij}(I_{ij})) = <I_{ij}, f_{ij}, O_i>$

$sr(O_i = f_{ij}(I_{ij})).I = I_{ij}$

$sr(O_i = f_{ij}(I_{ij})).f = f_{ij}$

$sr(O_i = f_{ij}(I_{ij})).O = O_i$

A Functional Assertion is a generalization of the set of information characterizable by a Hoare-logics expression. Thus, the semantic value of a Functional Assertion is equal to just this set of information.

$sv(O_i = f_{ij}(I_{ij})) \equiv someP(I_{ij})\ \{\ f_{ij}\ \}\ someR(I_{ij}, O_i)$

In order to fully specify a Functional Assertion, all of the information required by the semantics of such a P() {Q} R() expression must be included.

As an example :

Let *sv(x) = distance in "kilometers"*
$sr(x)$ = REAL with precision 1×10^{-2}
$sv(y)$ = time in "hours:minutes:seconds"
$sr(y)$ = time_structure = record
hours : INTEGER
minutes : INTEGER
seconds : INTEGER
end record
$sv(z)$ = velocity in "kph" <= 500 kph
$sr(z)$ = REAL with precision 1×10^{-2}
$sv(f_{ij})$ = "kph = "kilometers" + "hours"
$sr(f_{ij})$ = positive REAL = positive REAL + time_structure
$z \equiv O_i$
$x \in I_{ij}$
$y \in I_{ij}$

Then

$(sv(O_i = f_{ij}(I_{ij}))$
$\equiv$
$sv(z = f_{ij}(x,y))$
$\equiv$
$someP(I_{ij}, O_i)\ \ \{f_{ij}\}\ \ someR(I_{ij}, O_i)$

$\equiv$

$((0 \text{ kilometers} = sv(x)) \wedge (sr(x) = \text{REAL, precision } 1 \times 10^{-2})$
$\wedge$
$(0 \text{ hours} < sv(y))$
$\wedge$
$(sr(y) = \text{time\_structure})$
$\wedge$
$(sr(z) = \text{REAL with precision } 1 \times 10^{-2}))$

$\{f_{ij}\}$

$[x', y', z', f_{ij}' \mid$
$(sv(f_{ij}') = sv(f_{ij}))$
$\wedge$
$(sv(x') = sv(x))$
$\wedge$
$(sv(y') = sv(y))$
$\wedge$
$(sr(f_{ij}') = sr(z') = sr(x') +' sr(y'))$
$\wedge$
$(((sv(x') +' sv(y') = 500 \text{ kph})$
$\wedge$
$(sv(z') = sv(x') +' sv(y')))$
$\vee$
$(sv(z') = 500 \text{ kph})))$
$\wedge$
$(sv(z) = sv(z'))$
$\wedge$
$(sr(z) = \text{REAL with precision } 1 \times 10^{-2})]$

A point to note is the introduction of x', y', z', f_{ij}', and +' in the postcondition. This captures within the specification the realization that the division implied by $sv(f_{ij})$ does not make mathematical sense. The specification of a Functional Assertion should capture the fact that the objects x, y, and z are of different sr()'s and must be (in fact are required to be due to the semantic properties of the "division" operator in most strongly typed HOL's) converted to a common sr() prior to their actual mathematical division. At the same time, it indicates the fact that sr(x), sr(y), and sr(z), are data types of the true input and output objects and that it is an a-priori conclusion their syntactic representations are inviolate. However, the specification should not limit design freedom by stating what the modified sr()'s should actually be.

Each given output object for a real-time embedded system can be, and generally is, the focus of more than one functional mapping. RESS theory defines a relationship between a system's required functional mappings and "when" those mappings are required to be valid during the run-time operation. The attribute "when", within the domain of requirement specifications, maps a Functional Assertion to the point(s) during run-time at which it is *relevant to the overall* specification. [14]

The manner by which time is modeled is not as important as the need to model time using some appropriately defined temporal logic. Indirect approaches to modeling real-time aspects of embedded systems such as those based solely on finite state transition specifications, such as the *Control Flow model of* Hatley-Pirbhai [15], are inadequate for specifying many temporal aspects of RES's. It is difficult to specify important concepts such as "maximum data transport delay", "frequency of data object transmission", "maximum allowable time of loss of input object receipt", within such models; at least within the formal syntax of the model. It is also difficult to "reason" about those temporal aspects which they do model. Such specification attributes as temporal consistency and temporal completeness are not defined for such approaches.

RESS theory generalizes the concept of finite state to that of timepoint and interval (a bounded sequence of timepoints). This allows system "real-time" to be included in the model and defines a state transition as either the lower-bound or upper-bound of an interval.

RESS theory proposes that each Functional Assertion must have associated with it a description of "when" it is relevant. RESS theory elevates this temporal perspective to the level of an axiom, the Axiom of Temporal Relevance. This binding between a functional, or semantic, component of a specification and its associated temporal, or syntactic (in the sense of constraints on the organization of the semantic properties) component is embodied within a "Temporal Relevance Assertion".

Definition : Temporal Relevance Assertion D9

A Temporal Relevance Assertion, $T_{ij}(O_i=f_{ij}(I_{ij}))$, is a meta-relation consisting of one or more second order sub-assertions constraining the valid points on the temporal axis of the combined input/output state space. The i'th j'th Temporal Relevance Assertion defines the temporal conditions *required to exist in order for the j'th* Functional Assertion producing output object O_i to be considered a true relationship and, thus, relevant to the overall specification of the software.

In other words, a Temporal Relevance Assertion defines the time points during run-time at which the specification of the Functional Assertion which is its only parameter, is relevant to the *correct operation of the software* system.

The semantic value of a Temporal Relevance Assertion is a set of time points at which the Functional Assertion, which is its only parameter, is a true relation between its input and output state space parameters; according to the following rules :

sv(T(...)) = a single time point τ : $[0 <= \tau]$

or a single set of sequential time points $\tau_{\alpha\ldots\beta}$ called a well-formed, wf, interval :

$$(0 <= \tau_\alpha < \tau_\beta) \wedge \forall \tau_i \in \tau_{\alpha\ldots\beta} : [(\tau_\alpha <= \tau_i <= \tau_{i+1} <= \tau_\beta)]$$

or a wf set of non-sequential time points τ_i :

$$\forall i,j : [(0 <= \tau_i) \wedge ((i \neq j) \dashrightarrow (\tau_i \neq \tau_j))]$$

or a wf set of wf intervals $\tau_{\alpha i\ldots\beta i}$:

$$\forall i,j : [(0 <= \tau_{\alpha i} < \tau_{\beta i}) \wedge ((j \neq i) \rightarrow \neg ((\tau_{\alpha i} <= \tau_{\alpha j} <= \tau_{\beta i}) \vee (\tau_{\alpha i} <= \tau_{\beta j} <= \tau_{\beta i})))]$$

or a wf set of wf sets of non-sequential time points and wf intervals.

$T_{ij}(O_i = f_{ij}(I_{ij}))$

$<-->$

$O_i = f_{ij}(I_{ij})$ @time rtc = 0

or discrete event;

$T_{ij}(O_i = f_{ij}(I_{ij}))$

$<-->$

$O_i = f_{ij}(I_{ij})$ @time 50 ms. after $(O_{(i-1)} = f_{(i-1)k}(I_{(i-1)k}))$

or indirectly defined by a set of changes in the truth value of some modal, finite-state-defining, predicate;

$T_{ij}(O_i = f_{ij}(I_{ij}))$

$<-->$

$O_i = f_{ij}(I_{ij})$ while $\tau_{\alpha \ldots \beta}$
$\wedge$
$\exists\, I_m \subset I - I_{ij}$: [$\neg$ modal_predicate(I_m) @time $\tau_{\alpha-1}$
$\wedge$
modal_predicate(I_m) @time τ_α
$\wedge$
modal_predicate(I_m) @time τ_β
$\wedge$
$\neg$ modal_predicate(I_m) @time $\tau_{\beta+1}$]

The syntactic representation of a Temporal Relevance Assertion is the manner by which the time points of the semantic value are indicated. The above examples made use of temporal operators such as "@time", "after", and "while". They have semantic interpretations similar to those described in Kroger's Temporal Logic of Programs [16]. Other approaches, such as Ladkin's interval calculus [17], are also applicable.

Within RESS theory the *concept* of Functional Requirement may now be defined.

Definition : Functional Requirement D10

A Functional Requirement, $fReq_i(I_i,O_i)$, is the set of all Temporal Relevance Assertions $T_{ij}(O_i=f_{ij}(I_{ij}))$ having O_i as a common output state space.

More formally :

$\forall\, i,j$: [$T_{ij}(O_i=f_{ij}(I_{ij}))$
$->$
$T_{ij}(O_i=f_{ij}(I_{ij})) \in fReq_i(I_i,O_i)$)]

This definition for Functional Requirement constrasts sharply with the traditional definition. A more traditional definition emphasizes the concept of "Function" as the focus of a requirement, rather the output object which is required to have the functional relationship wrt. to a set of input objects. This is a result of a process-oriented, an example of "how", view to requirement specifications where a given output object does not exist until "produced" by the function.

Within RESS theory, the concept of Functional Requirement Specification has the following definition.

Definition : Functional Requirement Specification D11

Functional Requirement Specification, $fRSpec(I,O)$, is the set of all Functional Requirements $fReq_i(I_i,O_i)$ for the software specification abstraction level having I as its input state space and O as its output state space such that :

$\forall\, i$: [($fReq_i(I_i,O_i) \wedge (I_i \subset I) \wedge (O_i \subset O)$)
$->$
($fReq_i(I_i,O_i) \in fRSpec(I,O)$)]

In addition to separating Functional Requirement Specifications along one syntactic/semantic duality, Temporal Relevance Assertions and Functional Assertions, RESS theory also separates requirement information along another such duality; that of Non-Functional Requirement Specifications and Functional Requirement Specifications.

RESS theory considers such attributes as "maintainable", "understandable", "memory size efficient", etc. to fall into the category of Non-Functional Requirement. Such attributes are difficult to quantify due to their subjective non-discrete nature. This suggests a Fuzzy-Set-Theoretic approach for the specification and analysis of Non-Functional Requirements. The "fuzzy-like" nature of Non-Functional Requirements is an accepted truism. Thus, a "level of priority" attribute is associated with the Non-Functional Requirement within the definition of Non-Functional Requirement Assertion.

Definition : Non-Functional Requirement Assertion D12

A Non-Functional Requirement Assertion, $nfReq_{ij}(I_i,O_i)$, is the j'th assertion stating a required attribute of, or constraint governing, the design intended to satisfy $fReq_i(I_i,O_i)$.

RESS theory treats Non-Functional Requirements to be constant with respect to the temporal axis of the system's state space. Thus, there is no temporal assertion binding with each Non-Functional Assertion as for the Functional Assertion. Such a temporal attribute would imply software self-modification, as will be seen later in the discussion of software design. While this is a computational possibility, it does not normally fall in the domain of real-time embedded systems and will be consequently ruled out.

Definition : Non-Functional Requirement Specification D13

A Non-Functional Requirement Specification. $nfRSpec(I,O)$, is the set of all $nfReq_{ij}(I_i,O_i)$ for all $O_i \subset O$ associated with the design of a given abstraction level, <I,O> of a software system.

Software Design Specifications in RESS Theory

RESS theory identifies a separation of software design specifications along its semantic/syntactic boundaries. This separation has a natural isomorphic relationship to the Functional vs. Non-Functional Requirement Specification duality at the same abstraction level. RESS theory defines Data Transformation Design as that specification which defines a partial ordering of data-object-specific transformations whose syntactic and semantic composition satisfies some Functional Requirement from the set of Functional *Requirements making up the* $fRSpec()$ for that level of abstraction.

In addition, RESS theory defines Architectural Design as that specification that defines a data-type-specific partial ordering and modular packaging of Data Transformers. It also specifies a temporal binding of specific data objects to a Data Transformer partial order which is logically complete and consistent with respect to the set of Data Transformation Sequence specifications. The syntactic organization specified by Architectural Design has the goal of satisfying one or more Non-Functional Requirements from the set of $nfReq_i(I_i,O_i)$ making up the $nfRSpec(I,O)$.

In order to describe and reason about these two aspects of software design a number of additional terms must be defined.

Definition : Canonical Data Transformation D14

A Canonical Data Transformation, $\Delta c()$, is the general term for the discrete manipulation of either the syntactic representation or semantic value of a data object, or set of data objects, to produce a new syntactic representation or semantic value, respectively.

Naturally, there are two canonical data transformation types; a syntactical transformation and a semantic transformation. A canonical transformation must be object specific and have a syntactic or semantic result which contributes to the syntax or semantics of the functional assertion it is intended to refine/satisfy. Canonical semantic transformations map to functional relations within the postconditional semantics associated with the functional assertion. From this view, a data transformation sequence is simply a decomposition of a requirement expression. However, this stage of specification requires that design choices for the syntactic representation of the data objects carrying semantic information between canonical semantic transformations, be made. Such data object design decisions prejudice the syntactic canonical transformations needed to set up the data objects for the "required" semantic transformations. This also implies the need for a partial ordering of the transformations which is the beginning of the attachment of "process orientation" to the design of the solution. The next two definitions describe the two classes of canonical transformation more formally.

Definition : Semantic Value Transformation D15

A Semantic Value Transformation, $x := \Delta sv(y_1 \ldots y_n)$, is an object specific n-ary operator that specifies the conversion of the semantic values of a set of data objects to a new semantic value which it then binds to a pre-existing data object according to the following rule :

Let x, and $y_1 \ldots y_n$ each denote some data object

Then

$$
\begin{array}{l}
x := \Delta sv(y_1, \ldots y_n) \\
\quad \longleftrightarrow \\
\text{exists}(sr(x)) \\
\quad \wedge \\
\forall\, i \in \{1 \ldots n\} : [(sv(y_i) \neq sv(x))] \\
\quad \wedge \\
(sv(x) = sv(\Delta sv(y_1, \ldots y_n))
\end{array}
$$

Where

":=" has the traditional semantics "the values are equal after binding the results of the right hand side to the data object of the left hand side".

For example :

Let vel denote a velocity object
d denote a distance object
t denote a time object

$$
\begin{array}{ll}
\text{Then} & vel := \Delta sv(d,\ t) \\
& \quad \longleftrightarrow \\
& sv(\Delta sv) = \text{"+"} \equiv \text{idealized real number "division"} \\
& \quad \wedge \\
& sr(vel) = \text{an object of type REAL} \\
& \quad \wedge \\
& sv(vel) = \text{"100.0 kph"} \\
& \quad \wedge \\
& sr(d) = \text{an object of type REAL} \\
& \quad \wedge \\
& sv(d) = \text{"100.0 km"} \\
& \quad \wedge \\
& sr(t) = \text{an object of type REAL} \\
& \quad \wedge \\
& sv(t) = \text{"1.0 hours"}
\end{array}
$$

Definition : Syntactic Conversion Transformation D16

A Syntactic Conversion Transformation, $x := \Delta sr(y_1, \ldots y_n)$, is an object specific n-ary operator which converts or combines the syntactic representation *of elements of* its *domain* to a single, possibly *non*-primitive, element in its codomain. Such transformations conform to the following rules.

Let x,y each denote some data object

Then

$$
\begin{array}{l}
x := \Delta sr(y) \\
\quad \longleftrightarrow \\
(sr(y) \neq sr(x)) \\
\quad \wedge \\
(sv(y) = sv(x)) \\
\quad \wedge \\
(sv(\Delta sr) = f(): sr(x) \rightarrow sr(y) \\
\qquad\qquad\qquad\quad sv(x) \rightarrow sv(x))
\end{array}
$$

For example :

Let vel denote a velocity object
vel' denote an additional velocity object

$$
\begin{array}{ll}
\text{Then} & vel' := \Delta sr(vel) \\
& \quad \longleftrightarrow \\
& sv(\Delta sr) = \text{REAL\_to\_INTEGER conversion} \\
& \quad \wedge \\
& sr(vel) = \text{an object of type REAL} \\
& \quad \wedge \\
& sv(vel) = \text{"100.0 kph"} \\
& \quad \wedge
\end{array}
$$

$sr(vel') = $ an object of type INTEGER
$\wedge$
$sv(vel') = $ "100 kph"

RESS theory requires for each Functional Assertion one or more sequences of Data Transformations such that the semantic result of each Data Transformation Sequence is equal to the semantic relationship denoted by the Functional Assertion. Also the syntactic result of each sequence is equivalent to the output state space of the Functional Assertion. The number of such sequences is determined by the number of mutually exclusive functional relations present in the postconditional semantics of the Functional Assertion.

Definition : Data Transformation Sequence D17

A Data Transformation Sequence, $\Delta Seq_{ij}(I_{ij},O_i)$, is a partially ordered set of Canonical Data Transformations, $\Delta sv()$'s and $\Delta sr()$'s, such that :

$$sv(\Delta Seq_{ij}(I_{ij},O_i)) \equiv someP(I_{ij}) \{ \Delta Seq_{ij} \} someR(I_{ij},O_i)$$
$$= sv(O_i = f_{ij}(I_{ij}))$$

$$sr(\Delta Seq_{ij}(I_{ij},O_i)).O = O_i = sr(O_i=f_{ij}(I_{ij})).O$$
$$sr(\Delta Seq_{ij}(I_{ij},O_i)).I = I_{ij} = sr(O_i=f_{ij}(I_{ij})).I$$
$$sr(\Delta Seq_{ij}(I_{ij},O_i)).\Delta = \Delta Seq_{ij} = \text{partial order of } n\ \Delta c()\text{'s}$$

The semantics of a given ΔSeq_{ij} may be said to be equal to the semantics of its associated Functional Assertion only under the following condition.

$$sv(\Delta Seq_{ij}(I_{ij},O_i)) = sv(O_i = f_{ij}(I_{ij}))$$
$$<->$$
$$is\_well\_formed(sr(\Delta seq_{ij}(I_{ij},O_i)).\Delta)\_wrt.\_O_i = f_{ij}(I_{ij})$$

The symbol "wrt." stands for "with respect to".

Definition : ()_is_well_formed_wrt._() D18

The definition of the predicate "_is_well_formed_wrt._" is an inductive/recursive one according to the following rules :

1. $(x := \Delta c(y_1,...y_n))\_is\_well\_formed\_wrt.\_O_i = f_{ij}(I_{ij})$

$<->$

$\Sigma_{k=1}\ x_k = O_i$
$\wedge$
$\Sigma_{k=1...n}\ y_k = I_{ij}$
$\wedge$
$sv(\Delta c) = sv(f_{ij})$

where Δc is some Δsv or Δsr

2. $(x := \Delta c_1(\Delta c_2(y_1,...y_n),...\Delta c_m(w_1,...w_1)\_is\_well\_formed\_$

$\_wrt.\_(O_i = f_{ij}(I_{ij}))$

$<-->$

$(x := \Delta c_1(z_2,...z_m))\_is\_well\_formed\_$

$\_wrt.\_(O_i = f_{ij1}(\Sigma_{k=2...m}\ z_k)))$

$\wedge$

$\forall q \in \{2...m\} : [(z_q := \Delta c_q(I_{ij}))\_is\_well\_formed$

$\_wrt.\_(z_q = f_{ijq}(I_{ij}))]$

As a simple example :

```
Let z      be an output data object
    sr(z) = REAL
    sv(z) = kph
    x      be an input object
    sr(x) = INTEGER
    sv(x) = kilometers
    y      be an input object
    sr(y) = ARRAY [0..3] of CHAR
    sv(y) = hundredths of hours
    ÷      be a REAL number division operator

Then

    z := create_REAL_object()
    z := ÷(convert_INTEGER_to_REAL(x),
           convert_ARRAYofCHAR_to_REAL(y))
```

The Data Transformation Sequence also serves as the starting point for a formal definition of requirement satisfaction traceability within RESS theory.

Definition : ()_is_F-satisfied_by_() D19

This predicate, _is_F-satisfied_by_, states the condition under which a given Functional Assertion may be defined as being Functionally satisfied by a given Data Transformation Sequence.

$(O_i = f_{ij}(I_{ij}))$_is_F-satisfied_by_$(\Delta Seq_{ij}(I_{ij},O_i))$

<-->

$sv(\Delta Seq_{ij}(I_{ij},O_i)) = sv(O_i = f_{ij}(I_{ij}))$

RESS theory proposes the specification classes Data Transformation Set, and Data Transformation Specification in order to provide terms for the grouping of all transformation sequences which result in the creation and semantic binding of the same output data object or same embedded system abstraction level.

Definition : Data Transformation Set D20

A Data Transformation Set, $\Delta Set_{ij}(I_{ij},O_i)$, is the set of all Data Transformation sequences $\Delta seq_{ij}(I_{ij},O_i)$ having O_i as its associated output state space and $O_i = f_{ij}(I_{ij})$ as its Functional Assertion.

Definition : Data Transformation Specification D21

A Data Transformation Specification $\Delta Spec(I,O)$ is the set of all Data Transformation Sets in the following manner:

$\forall\, O_i \subset O : [\Delta Set_{ij}(I_{ij},O_i) \rightarrow \Delta Set_{ij}(I_{ij},O_i) \in \Delta Spec(I,O)]$

RESS theory delegates the "fuzzy"-satisfaction of Non-Functional Requirements to the specification class "Architectural Design." The design process resulting in the specification of this class has the goal of satisfying such difficult to quantify requirements like "minimum possible memory space", "easy to maintain", "minimum inter-modular coupling", etc. The primitive in this specification class is the Data Transformer.

Definition : Data Transformer D22

A Data Transformer, $\Delta Dt_g(I_g,O_g)$, is a generalization of the semantic and syntactic attributes of one or more total or partial Data Transformation Sequences. Unlike a $\Delta Seq()$, which is data object specific, a $\Delta Dt()$ is data object generic but data type specific.

$sv(\Delta Dt_g(I_g,O_g)) \equiv someP(I_g)\ \{\ \Delta Dt_g\ \}\ someR(I_g,O_g)$

$sr(\Delta Dt_g(I_g,O_g)).I = I_g$

$sr(\Delta Dt_g(I_g,O_g)).O = O_g$

$sr(\Delta Dt_g(I_g,O_g)).\Delta = \Delta Dt_g$

A Data Transformer is a software design element which groups subsets of ΔSeq() partial orders in a common packages for the purpose of "fuzzy"-satisfying "real estate oriented" Non-Functional Requirements; such as the amount of memory space or temporal axis length available for carrying out a given function. A "fuzzy" predicate relationship describing the degree of satisfaction of a given nfReq() will be investigated as an addition to RESS theory. Such a relation will permit each nfReq() to be traced to those ΔDt()'s which are involved in its satisfaction.

Data Transformers are an abstraction of such implementation specification concepts as "software system", at the highest level of abstraction, "task" and "procedure", at intermediate levels, and "statement" and "instruction" at low levels of abstraction.

Definition : Data Transformer Sequence D23

The Data Transformer Sequence, ΔDtSeq(), is a partial order of ΔDt()'s which, when bound to specific subsets $\langle I_{ij}, O_i \rangle \subset \langle I, O \rangle$ subsume the state spaces and semantics of one or more $\Delta Seq(I_{ij}, O_i)$'s and the Temporal Relevance Assertions governing the Functional Assertions satisfied by each $\Delta Seq(I_{ij}, O_i)$.

$$sv(\Delta DtSeq_{ij}(I_{ij}, O_i)) \equiv someP(I_{ij}) \wedge RelevanceP(I_{ij}, O_i)$$

$$\{ \Delta DtSeq_{ij} \}$$

$$someR(I_{ij}, O_i)$$

$$sr(\Delta DtSeq_{ij}(I_{ij}, O_i)).I = I_{ij}$$

$$sr(\Delta DtSeq_{ij}(I_{ij}, O_i)).O = O_{ij}$$

$$sr(sv(\Delta DtSeq_{ij}(I_{ij}, O_i))).P = someP(I_{ij})$$

$$sr(sv(\Delta DtSeq_{ij}(I_{ij}, O_i))).Rp = RelevanceP(I_{ij})$$

$$sr(sv(\Delta DtSeq_{ij}(I_{ij}, O_i))).R = someR(I_{ij})$$

The Data Transformer Sequence constitutes the second and third points at which requirement satisfaction traceability relations are recognized within RESS theory.

Definition : ()_is_A-satisfied_by_() D24

The predicate, _is_A-satisfied_by_, states the condition under which the semantics of a given ΔSeq() may be defined as being Architecturally satisfied by a given ΔDtSeq().

$$(\Delta Seq_i j(I_{ij}, O_i))\_is\_A\text{-}satisfied\_by\_(\Delta DtSeq_{kl}(I_k{}^1, O_k))$$

$$\longleftrightarrow$$

$$sr(\Delta Seq_{ij}(I_{ij}, O_i)).I \subset sr(\Delta DtSeq_{kl}(I_{kl}, O_k)).I$$
$$\wedge$$
$$sr(\Delta Seq_{ij}(I_{ij}, O_i)).O \subset sr(\Delta DtSeq_{kl}(I_{kl}, O_k)).O$$
$$\wedge$$
$$sv(\Delta Seq_{ij}(I_{ij}, O_i)) \rightarrow sv(\Delta DtSeq_{kl}(I_{kl}, O_k))$$

Definition : ()_is_T-satisfied_by_() D25

The predicate, _is_T-satisfied_by_(), states the condition under which the temporal semantics of $T_{ij}(O_i = f_{ij}(I_{ij}))$ are satisfied by a relevance predicate which, as part of the precondition of a given $\Delta DtSeq_{kl}(I_{kl}, O_k)$, is a refinement of some component of $T_{ij}(O_i = f_{ij}(I_{ij}))$.

$$(T_{ij}(O_i = f_{ij}(I_{ij})))\_is\_T\text{-}satisfied\_by\_(\Delta DtSeq_{kl}(I_k{}^1, O_k))$$

$$\longleftrightarrow$$

$$sr(T_{ij}(O_i = f_{ij}(I_{ij}))).I \subset sr(\Delta DtSeq_{kl}(I_{kl}, O_k)).I$$
$$\wedge$$
$$sr(T_{ij}(O_i = f_{ij}(I_{ij}))).O \subset sr(\Delta DtSeq_{kl}(I_{kl}, O_k)).O$$
$$\wedge$$
$$sv(T_{ij}(O_i = f_{ij}(I_{ij}))) \rightarrow sr(sv(\Delta DtSeq_{kl}(I_{kl}, O_k))).Rp$$

Thus, the connections between Data Transformers that form the Data Transformer Sequences may be considered an abstraction of the implementation specification concepts of "task invocation", "procedure call", "sequence", and "concurrence". Each Data Transformer Sequence specifies a binding of data-object-independent Data Transformers to specific data objects. The result is an execution path from the input state space of the abstraction level to the output state space. This execution path through Data Transformers must be chosen to satisfy the semantic and syntactic requirements specified by one or more ΔSeq()'s under the temporal constraints specified by the $T_{ij}(O_i=f_{ij}(I_{ij}))$'s associated with the $O_i=f_{ij}(I_{ij})$'s whose functional semantics and syntax are satisfied by the ΔSeq()'s. In essence it is a re-connection of the partial *ΔSeq()'s within the* individual Data Transformers such that the set of Data Transformation Sequences is recreated.

Definition : Architectural Design Package D26

An Architectural Design Package, $Adp_i(I,O)$, is the i'th set of all ΔDt()'s whose grouping together act to "fuzzy"-satisfy such qualitative Non-Functional Requirement types as "Easily modifiable", High degree of reusability", etc. for the Specification Abstraction Level associated with <I,O>.

An Architectural Design Package has no function other than to address Non-Functional Requirements. In this attribute it is an abstraction of implementation specification elements such as an Ada package or Pascal module.

Definition : Architectural Design Specification D27

An Architectural Design Specification, Ads(I,O) is the set of all Architectural Design Packages for a given Specification Abstraction Level.

Now that the above terms have been been defined, "System", "Embedded System", and "Real-Time Embedded System" can be defined within the domain of RESS theory.

Definition : System

A System, S(I,O), is a single Data Transformer with data type specific, but data object independent, input state space sr(I) and output state space sr(O).

Definition : Embedded System D28

An Embedded System, ES(I,O), is an S(I,O) in which the input state space sr(I) and the output state space sr(O) is finitely bound to specific data objects and for which there exists a deterministic response state sv(O) within the domain of the output state space sr(O) for any stimulus state sv(I) possible within the domain of the input state space sr(I).

Definition : Real-Time Embedded System D29

A Real-Time Embedded System, RES(I,O), is an ES(I,O) in which "time" is a member of the input state space sr(I) with one or more of the output data objects $sr(O_i) \subset sr(O)$ dependent on a relation between the input data objects $sr(I_{ij}) \subset sr(I)$ of its $\Delta Seq(I_{ij},O_i)$ and "time".

Basic Axioms of RESS Theory

The following three axioms, along with the above definitions, form the basis for RESS theory. These axioms were first discussed in different forms in [18], [19].

Axiom of System Hierarchy A1

Any given RES(I,O) is a single Data Transformer within a Data Transformer Sequence of a higher level of Specification Abstraction.

The implication of A1 is that for any given RES(I,O), the specifications for sr(I), sv(I), sr(O), sv(O), Precondition(I), Relevance Predicate(I,O), and Postcondition(I,O) are a-priori inheritable from the Architectural Design Specifications of a higher level system.

Axiom of Deterministic Response A2

A semantic value of the output state space of an RES(I,O) is specifiable for any semantic value of its input state space.

As a corollary : for any given RES(I,O),

wp(RES(I,O), is_valid(O)) = true wrt. I

where wp() is Dijkstra's weakest-precondition predicate transformer. [20]

Axiom of Temporal Relevance A3

The Functional Assertions required to be satisfied by the design of an RES(I,O) are relevant to the Data Transformation Sequences of its "parent" system only at specifiably discrete points in time.

Theorems of RESS Theory

The following is a partial list of theorems of RESS theory. They are included to provide a overview of some of the additional attributes of RESS-theoretic specifications that are not immediately noticeable from the above definitions and discussions. Concepts such as "sv consistency" and "sv completeness" for state spaces are left to their intuitive definitions. T5 through T10 and T12 are expanded in order to give a detailed picture of temporal and function completeness and consistency descriptions.

Theorem of State Space "sr" Completeness T1

ForAll RES(I,O), sr(I) and sr(O) are completely specifiable.

Theorem of State Space "sr" Consistency T2

ForAll RES(I,O), sr(I) and sr(O) are consistently specifiable.

Theorem of State Space "sv" Completeness T3

ForAll RES(I,O), sv(I) and sv(O) are completely specifiable.

Theorem of State Space "sv" Consistency T4

ForAll RES(I,O), sv(I) and sv(O) are consistently specifiable.

Theorem of Functional Requirement Temporal Completeness T5

A Requirement Specification is semantic-temporally complete if ForAll data objects of the output state space the required sv(data object) is defined by a Functional Assertion for all timepoints beginning at the object's own $\tau=0$ (established upon its initialization).

In other words :

$\forall$ *f*RSpec(I,O) : [Is_sv_Temporally_Complete(*f*RSpec(I,O))

<-->

$\forall O_i \in O$

$\forall \tau \in \tau_{0 \ldots \infty}$

$\exists T_{ij}(O_i = f_{ij}(I_{ij})) : [\tau \in sv(T_{ij}(O_i = f_{ij}(I_{ij})))\ ]$

Theorem of Functional Requirement Temporal Consistency T6

A Requirement Specification is semantic-temporally consistent if ForAll data objects of the output state space the sv(data object) is not defined by more than one functional assertion at the same time point on the temporal axis.

In other words :

$\forall$ *f*RSpec(I,O) : [Is_sv_Temporally_Consistent(*f*RSpec(I,O))

<-->

$\forall\ O_i\ \varepsilon\ O$

$\exists\ fReq_i(I_i, O_i)$:

$\forall\ i,\ j,\ k$:
$[((T_{ij}(O_i = f_{ij}(I_{ij}))\ \varepsilon\ fReq_i(I_i, O_i))$
$\wedge$
$(T_{ik}(O_i = f_{ik}(I_{ij}))\ \ \varepsilon\ fReq_i(I_i, O_i))$
$\wedge$
$(T_{ij}(O_i = f_{ij}(I_{ij}))\ \neq\ T_{ik}(O_i = f_{ik}(I_{ik}))\ )$

$\dashrightarrow$

$(sv(T_{ij}(O_i = f_{ij}(I_{ij})))\ \cup\ sv(T_{ik}(O_i = f_{ik}(I_{ik}))) = \{\phi\})]$

Theorem of Functional Requirement "sr" Completene< **T7**

ForAll RES(I,O)
ForAll timepoints $\tau_i \geq$ @time $\tau_i = 0$
ForAll $sr(O_i) \subset sr(O)$
ThereExists a syntactic relational mapping
$sr(O_i) = sr(f_{ij})(sr(I_{ij}))\ \ : [sr(I_{ij}) \subset sr(I)]$

Theorem of Functional Requirement "sr" Consistency **T8**

ForAll RES(I,O)
ForAll i,j,k,l :
$[(sr(O_i) = sr(f_{ij})(sr(I_{ij}))$
$\wedge$
$sr(O_l) = sr(f_{lk})(sr(I_{lk})))$
$\wedge$
$sr(O_i) = sr(O_l)$
$\wedge$
$sr(I_{ij}) = sr(I_{lk}))$

$\rightarrow$

$sr(f_{ij}) = sr(_{lk})\ ]$

Theorem of Functional Requirement "sv" Completeness **T9**

ForAll RES(I,O)
ForAll timepoints $\tau_i \geq$ @time $\tau_i = 0$
ForAll $sr(O_i) \subset sr(O)$
ThereExists a semantic relational mapping
$sr(O_i) = sr(f_{ij})(sr(I_{ij}))\ \ : [sr(I_{ij}) \subset sr(I)]$

Theorem of Functional Requirement "sv" Consistency **T10**

ForAll RES(I,O)
ForAll i,j,k,l :
$[(sv(O_i) = sv(f_{ij})(sv(I_{ij}))$
$\wedge$
$sv(O_l) = sv(f_{lk})(sv(I_{lk})))$
$\wedge$
$sv(O_i) = sv(O_l)$
$\wedge$
$sv(I_{ij}) = sv(I_{lk}))$

$\rightarrow$

$sv(f_{ij}) = sv(_{lk})\ ]$

Theorem of Functional Relation Classification **T11**

Each Functional Relationship present in the postcondition semantics of a Functional Assertion may be classified as, or decomposed into, either "semantic" or "syntactic" with respect to the $sr(O_i)$ and $sv(I_{ij})$ overwhich the relationship is required to hold.

Theorem of Data Transformation Sequence **T12**

ThereExists a Data Transformation Specification $\Delta Seq(I_{ij}, O_i)$ for all mutually exclusive Functional Relations f_{ijk} ForAll Functional Assertions $O_i = f_{ij}(I_{ij})$ of all $fReq(I_i, O_i)$ in the $fRSpec(I,O)$ of a given abstraction level for any embedded system RES(I,O).

Theorem of How/What Duality T13

For any given Specification Abstraction Level, the complete set of Functional Assertion Specifications, the "what" for that level, are logically equivalent to the Data Transformer postcondition specifications, part of the "how" of the previous Specification Abstraction Level.

Theorems of Architectural Design Derivation T14.1

For any given Specification Abstraction Level, the Data Transformation Sequence specifications for that level are logically implied by the Functional Assertion specifications and "fuzzy"-logically implied by the Non-Functional Requirements for that same level.

T14.2

For any given Specification Abstraction Level, the Relevance Predicate governing the binding of input and output data objects with Data Transformers is logically implied by the Temporal Relevance Assertions constraining the associated Functional Assertions.

An Application of RESS Theory

The intended application of RESS theory is as a deductive inference model upon which to base an AI-enhanced Computer-Aided Specification Engineering, CASE, environment. The axioms, theorem, definitions, and associated rules of inference, when expressed in mathematical logic, enables the application of an inference engine approach to specification collection and analysis. Simplistically, the entry of specification information can be used to trigger a forward chain of reasoning causing the correct forms of analysis of the new information with respect to the model and the previous specification contents of the knowledge base. If, during such analysis, required specification information is found to be missing from the knowledge base a goal directed, ie., backward reasoning, process can be triggered prompting the collection of the missing information. A complete and consistent set of specification information will be provable as a theorem of RESS theory.

As stated earlier, current generation CASE tools have very little real understanding of the specification information they are responsible for collecting and "analyzing". They act as slaves to the user allowing the creation of graphic language specifications according to very simple syntactic rules without enforcing any semantic constraints on the information specified. They do not have the capability of assisting the user in semantic completeness or consistency analysis. Nor, are they capable of requesting or suggesting specific information which would correct completeness and/or consistency problems.

An AI-enhanced CASE environment, based on the model proposed by RESS theory, should be capable of performing the above analysis types [21, 22, 23]. It should also be capable of assisting in the selection of design choices based on the heuristics which all competent software engineers apply instinctively to their software designs. These capabilities, and others such as automatic code generation and correctness proofs, provides the software engineer with some of the tools now available to the digital hardware engineer via CAD systems. Current day CAD systems, based on internally manipulated knowledge of the semantics of NAND gates and Boolean algebra, for example, provide assistance to hardware designers only dreamed about in the software design domain.

Conclusion

At the time I submitted the abstract for this conference I was interested in expressing a number of concepts that I and a colleague had been working on related to the identification, analysis, specification, and application of requirements for real-time embedded systems. These concepts involved a hierarchical view of specification levels, A1, which is an extension of the traditional

$$\begin{tikzcd}
 & \text{System Requirements} \arrow[d] & \\
 & \text{System Design} \arrow[dl] \arrow[dr] & \\
\text{Hardware Requirements} & & \text{Software Requirements}
\end{tikzcd}$$

hierarchy but which generally falls into flatness at the hardware and software levels

$$\begin{tikzcd}
\text{Hardware Requirements} \arrow[d] & \text{Software Requirements} \arrow[d] \\
\text{Hardware Design} \arrow[d] & \text{Software Design} \arrow[d] \\
\text{Hardware Implementation} & \text{Software Implementation}
\end{tikzcd}$$

instead of recursively descending down successively refined abstraction levels within a software and hardware specification hierarchy. In addition, I felt that there was little work available identifying and formalizing the concepts of semantic completeness and semantic consistency for requirement specification and analysis. The same is true in regards to the semantic relationships between requirement specifications and design specifications (implementation being the least abstract level of design specifications). Semantics are generally discussed only in the context of source level programs and programming languages. The only such consistency and completeness attributes definable for most current approaches to requirement specification is syntactic in nature. For example, "level balancing" within the "structured analysis" camp.

Prior to this paper, the work we had done was primarily conceptual and experimental [24]. The axioms and theorems were expressed abstractly and were unproven. And the specification classes and their relationships, which were the focus of these axioms and theorems, were not formally defined. After beginning the paper I quickly found that the formal expression of the axioms, theorems, and rules of inference of RESS theory; which was my original intention; was not possible without first developing a vocabulary and set of logical operators with precise definitions. This would not be a surprise to a mathematician. However, as an electrical engineer currently working in the development of real-time avionics software, the amount of effort required to do just this much, was both unexpected and very fruitful.

This paper consists primarily of a first draft of a basic subset of this vocabulary and operator set. However, through the systematic presentation of this set, it does present a picture of the model of software specifications proposed by RESS theory. While this model is somewhat non-traditional, it hopefully raises a number of interesting and valid perspectives on the specification problem.

The emphasis on the semantic properties of specification and the temporal aspects of all real-time systems, beyond that of state transitions, attempts to bring into focus two areas of major shortcomings in most approaches to requirement specification and analysis. Partitioning of design specification classes into object specific Data Transformation Sequences, which has as its focus a system's Functional Requirements, and type specific Architectural Design, with a focus of "fuzzy"-satisfying the Non-Functional Requirements while subsuming the Data Transformation Sequences, proposes a separation of concerns which can assist the designer in both identifying, justifying and quantifying design decisions.

The application of a Fuzzy-Set-Theoretic approach to the specification, management, and application of Non-Functional Requirements shows a number of possible benefits and applications in the area of guiding design tradeoffs during the design selection and specification stages of a development.

The final goal, that of fully specifying and proving the soundness and completeness of RESS theory, is the subject of a continuing effort. In this light "RESS theory" should be referred to as "RESS hypothesis". An additional goal is the intended application of this theory as the deductive "proof-theoretic" model upon which an AI-based real-time embedded system specification assistant is developed and implemented. The vocabulary, definitions, and specification relationships developed during the work on this paper has definitely moved these goals a large step closer.

Acknowledgements

I would very much like to thank a friend and colleague, Mark Blackburn, currently an Allied/Signal Aerospace assignee with the Software Productivity Consortium, for introducing me to the works of E.W. Dijkstra, C.A.R. Hoare, and others on which the foundation for a true engineering approach to software development may be built.

References

1. J. Martin, "System Design from Provably Correct Constructs," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1985.

2. H.K. Berg, W.E. Boebert, W.R. Franta, T.G. Moher, "Formal Methods of Program Verification and Specification," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1985.

3. E.W. Dijkstra, "The Humble Programmer," Turing Award Lecture, cacm, vol. 15, oct 1972. pp. 859-866.

4. D. Teichroew, E.A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January, 1977.

5. A. Pizzarello, "Development and Maintenance of Large Software Systems," LLP, California, U.S.A., 1984.

6. A. Pizzarello, "WELLMADE: A Rational Design Methodology,", Internal Design Methodology Standard, Honeywell Information Systems, 1980.

7. D.J. Hatley, I.A. Pirbhai, "Strategies for Real-Time System Specification," Dorset House Publishing Co., Inc., New York, New York, U.S.A., 1987.

8. H.K. Berg, W.E. Boebert, W.R. Franta, T.G. Moher, "Formal Methods of Program Verification and Specification," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1985.

9. E.W. Dijkstra, "A Discipline of Programming," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1985.

10. C.A.R. Hoare, "An Axiomatic Basic for Computer Programming," Communications of the ACM, Vol. 12, No.10, Oct. 1969.

11. D. Gries, "The Science of Programming," Texts and Monographs in Computer Science, Springer-Verlag, New York, U.S.A., 1981.

12. M. B. Josephs, "An Introduction to the Theory of Specification and Refinement," IBM TJ Watson Research Center Report, Yorktown Heights, New York, U.S.A., 1987.

13. Z. Manna, R. Waldinger, "The Logical Basis for Computer Programming, Vol. 1: Deductive Reasoning," Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, U.S.A., 1985. pp. 189-190.

14. R.D. Busser, M.R. Blackburn, "Moving Structured Development Towards Proof of Correctness," Proceedings of Structured Development Forum VIII, Seattle, Washington, U.S.A., Aug. 1986.

15. D.J. Hatley, I.A. Pirbhai, "Strategies for Real-Time System Specification," Dorset House Publishing Co., Inc., New York, New York, U.S.A., 1987.

16. F. Kroger, "Temporal Logic of Programs," EATCS Monographs on Theoretical Computer Science, Spring-Verlag, New York, U.S.A., 1987.

17. P. Ladkin, "Specification of Time Dependencies and Synthesis of Concurrent Processes," Proceedings of the 9th International Conference on Software Engineering, Monterey, California, U.S.A., April 1987, pp. 106-115.

18. R.D. Busser, M.R. Blackburn, "Moving Structured Development Towards Proof of Correctness," Proceedings of Structured Development Forum VIII, Seattle, Washington, U.S.A., Aug. 1986.

19. M.R. Blackburn, "Automated Software Development Methodolgy (ASDM): An Architecture of a Knowledge-Based Expert System," Masters Thesis, Florida Atlantic University, Boca Raton, Florida, U.S.A, (Also Tech. Report SPC-TR-87-008, Software Productivity Consortium, Reston, Virginia, U.S.A.), 1987.

20. E.W. Dijkstra, "A Discipline of Programming," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1985.

21. M.R. Blackburn, R.D. Busser, J.C. Angermayer, "The Software Development Process: Past, Present, and a Proposal for the Future," Internal Report, Allied/Bendix Aerospace, Bendix Avionics Division, Fort Lauderdale, Florida, U.S.A., 1987.

22. R.D. BUSSER, "The ASDM System: A System Level Specification and Development Plan", Internal Report, Allied/Bendix Aerospace, Bendix Avionics Division, Fort Lauderdale, Florida, U.S.A., 1987.

23. D. Barstow, "Artificial Intelligence and Software Engineering", Proceedings of the 9th International Conference on Software Engineering, Monterey, California, U.S.A., April 1987, pp. 200-211.

DISCUSSION

K.Benner, US

What is the relationship between REFINE and Z?

Author's Reply

REFINE: REFINE is a tool capable of transforming a very high level language program, written in REFINE's own language "V", into progressively lower level programs, still in "V", until the level of the program is such that it is translatable from "V" directly into a target source code language. "V" incorporates set theoretic and proof theoretic constructs so it is possible to express declaratively, requirement and abstract design specifications. However, REFINE does not embody any inherent model of what such specifications should look like, what defines a complete and consistent set of specifications, or how to go about requesting such specifications from a user. These are the areas in which my work on RESS theory is concerned. REFINE acts like a compiler which a case tool base on the model proposed by RESS theory would act as an intelligent builder/collector of the class of information which would be suitable for expression as a "program" which REFINE would "compile".

Z: Z is a language useful for expressing specifications in a formal manner. RESS theory defines what information is required to be expressed formally in some language like "Z".

A.O.Ward, UK

How do you intend to educate the users of formal methods in your organization?

Author's Reply

I no longer believe it to be either possible or practical to try to educate currently employed software engineers in the required discrete maths and predicate calculus for application in formal specification methods. It requires a shift in the engineer's view of what mathematics is all about, from continuous mathematics to discrete. This conceptual shift seems to be very difficult or undesirable to most engineers. Instead, my efforts are now focused on the development of an axiomatic/deductive model which can be used as a foundation for an expert system-based case environment in which the "expertise" is that of a software engineer that does apply formal specification techniques. This approach buries the formal mathematics inside the case environment with the environment providing a more informal view of the information, such as data flow and control flow diagrams. The advantage of this approach is that the formal specification, built internally by the environment from an informal user view can then be analyzed for much more complete and rigorous definitions of attributes such as completeness and consistency (semantic as well as syntactic) based on the axiomatically characterized model.

A.O.Ward, UK

The REFINE tool is designed to describe the process of successive transformations in a formal context as well as the results of such steps and hence differs from mainstream formal methods such as Z and VDM. But like many such tools it lacks a method that will incrementally collect information in such a way as to encourage formal expression. Both methods *and* tools are required for successful application of formality.

Author's Reply

I agree, but would like to add that prior to "methods and tools" there must be an underlying theory which acts as the foundation of any basis for methods which must then be supported by appropriate tools. REFINE is a tool which is capable of taking a given set of formal specifications, when translated into "V", and transform them into a potential implementation at the source code level. The degree that this code is usable as actual application software, rather than prototype software, is a function of the constraints imposed by the non-functional requirements of the particular application. Also, the degree to which REFINE can incorporate user directives on priorities of non-functional requirement satisfaction and the heuristic rules necessary to guide the transformations selected for this satisfaction is a key to actual application code generation. If these capabilities can be made robust enough, REFINE could be a very valuable addition to a case environment capable of constructing formal specification suitable for translation into "V" for input to REFINE. This is, in fact, a direction my colleague, Mark Blackburn, is pursuing.

THE STATE OF PRACTICE IN ADA-BASED PROGRAM DESIGN LANGUAGES

Lawrence G. Jones
SHAPE Technical Centre
P.O. Box 174
22501 CD The Hague
The Netherlands

SUMMARY

The use of Ada in avionics and other systems is assuming more importance due to a growing body of national policies dictating its use in mission critical systems. This has created a need for software engineering techniques that incorporate Ada concepts so that the transition from a system design to an Ada implementation is more easily achieved. One of the most common techniques is to use an Ada-based program design language (ADL). This paper surveys the current state of practice in Ada-based program design languages by examining some leading reports on the subject. Among the more important efforts, we discuss the surveys of existing ADLs conducted for the US Naval Avionics Center (NAC), the Institute of Electrical and Electronics Engineers Recommended Practice for ADLs, the ADL guidelines produced for Transport Canada, and the ADL guidelines produced for the NAC. We note that the state of practice is considered to be too immature for standardisation. However, we summarise some of the major findings and point to future directions.

1. Introduction

The use of Ada in avionics and other systems is assuming more importance due to a growing body of national and international policies dictating its use in mission critical systems. For example, within NATO, Ada is the specified language for the 14 nation funded portion of the NATO Command, Control and Information System. Furthermore, because of the inherent advantages of Ada, many companies and national organisations are making Ada the language of choice even where the force of policy does not dictate its use.

While Ada has specific features to facilitate the design and implementation of systems, a programming language by itself cannot address all aspects of the software development life cycle. One of the most common techniques to bridge the gap between design and implementation is a program design language or PDL. When a PDL is Ada-based it is often referred to as an ADL. An ADL has potential utility for not only avionics but for all Ada implementations.

This paper will survey the state of practice in ADLs by examining the most influential recent reports on ADLs. This includes survey reports on existing ADLs, the results of an important standardisation attempt, and some important ADL guideline documents. Thus, the emphasis of this paper is not on specific ADLs, but rather on the state of ADLs as a whole.

We begin by presenting some background information to define an ADL and place it in the perspective of the software life cycle. We then examine the major product surveys, continue by discussing the standardisation attempt of the Institute of Electrical and Electronics Engineers (IEEE), and finish our coverage with a look at some major ADL guideline reports. We conclude by summarising major findings and pointing to the future.

2. Background

The story of Ada's development is certainly well known. Those wishing details on the history and features of Ada are referred to Booch (1). Briefly, Ada's development was initiated in 1975 by the US Department of Defense (DoD) as a common high order language for real-time embedded computer systems. Today, Ada is a mature language with wide international acceptance. The maturation of the language is represented by the fact that Ada is now a recognised standard by the US DoD (as a military standard (MIL-STD), the American National Standards Institute (ANSI), and the International Standards Organisation (ISO). The wide acceptance of the language is represented both by the standardisation and the requirement for using Ada in many national and international policies. There is official policy dictating the use of Ada for certain applications in the United States, Canada, the United Kingdom, West Germany, as well as within the Commission of European Communities and NATO. However, application of Ada to avionics is still relatively new. In 1984 the US Air Force F-15 flew trials with a flight-control computer programmed in Ada. Also in 1984, Northop claimed its F-20 was the first aircraft to fly with an operational mission computer programmed in Ada.

Given the maturity of the language itself, focus has shifted to the larger view of the system life cycle. People are now more concerned with how to incorporate Ada in the flow from project concept to implementation and beyond. Below, we take a brief look at the place of an ADL within this larger context.

In the classic view of the software system life cycle there are the following phases: Analysis, Design, Implementation, and Maintenance and Modification. This is often referred to as the "waterfall" life cycle model. In the Analysis Phase, the system requirements are identified and documented. The analysis documents specify "what" is to be done. In the Design Phase, software engineers design a solution that meets the requirements. The design documents consist of the engineering specification of "how" the requirements will be satisfied. For the software portion of a system, the Implementation Phase is where the working code is created. Maintenance and Modification of the system continue as mini-iterations of the overall life cycle as new requirements emerge. This continues until the system is retired.

There are many techniques available to address each of these life cycle phases. However, traditionally there have been problems in smoothly transitioning between phases. Often the tools and techniques appropriate for one phase may not be useful in another phase. A program design language (PDL) is a widely used tool to ease the transition between the Design and Implementation Phases. The classic reference on PDLs is Caine and Gordon (2). They declare that the purpose of PDL is threefold: to support evaluation of a design, to provide guidance to implementation, and to serve as a tool for maintenance. They propose that use of a PDL results in increased productivity for developers and improved system quality.

Initially, PDLs evolved from pseudocode. In form, pseudocode often looked like formalised program comments which used the three basic control structures (sequence, selection, and iteration) with English language narrative to indicate the operations performed. The designer would start the design process by using pseudocode to produce a "rough draft" of a program. Pseudocode supported the design principles of abstraction and stepwise refinement. It allowed the programmer to concentrate on high level abstractions and to avoid getting lost in the details of a programming language. Furthermore, the rough draft could be successively refined until a working program was produced. Thus, pseudocode was truly a bridge between design and implementation. Today a pseudocode that has been sufficiently formalised is usually referred to as a PDL and is a standard tool of the software engineer. While there is no universal agreement on all PDL features, most contain additional design information beyond the algorithmic information represented by pseudocode.

Almost from its introduction, Ada has been hailed as a design language. Indeed, there are many features that lend themselves to the development and expression of design. These features include strong support for abstraction of operations and data, representation of concurrent operation, specification of generic templates for subprograms, and convenient mechanisms for decision deferral. From the early days of the language, Ada programs were designed using Ada itself as a PDL, and today the term Ada-based PDL or ADL is commonplace. While there is no formally established definition of an ADL, one consistent with common practice is given here. An Ada-based Design Language may be defined as a textual, procedural design language that closely follows Ada syntax, at least supporting the detailed design phase, and usually extending the Ada language with constructs that are desirable for other design activities.

The state of practice regarding ADLs for Ada system development has followed the same rough pattern as previous software engineering practice. Initially, system developers were primarily concerned with the Implementation Phase of the life cycle. Only later did they work backwards toward the start of the life cycle to define a full life cycle methodology. While Ada designers did not forget the lessons of the past, some of the non-traditional features of Ada have called for new approaches. Thus, because many Ada development techniques have been built from the language backwards, it is not well established how you should produce designs to account for Ada's special features.

While the emphasis in this paper is on the ADL, many important future questions relate to the methodology preceding ADL in the life cycle. This issue will be briefly discussed at the end of the paper. We will continue now with a look at some of the major ADL studies.

3. ADL Surveys

The major leader in surveying the ADLs was the US Naval Avionics Center. In 1982, the Naval Avionics Center (NAC), awarded a contract to SoftTech to conduct a survey of existing ADLs (3). The survey found that ADL development was too immature to recommend any standards. They found only four true ADLs in use and found no consensus among them. The report concluded that, while standardisation was not appropriate, the Navy could promote the use of ADLs and the development of guidelines.

In 1984 the NAC recognised the increasing number of ADL efforts and reinitiated its survey project with a contract to Computer Technology Associates (4). The survey team collected initial information about 28 different PDLs. After initial analysis, the team determined that seven were in early developmental stages and were not appropriate for detailed evaluation. Five more were not sufficiently Ada-based. This left sixteen true ADLs that were eventually examined in detail.

To allow detailed comparison of the ADLs, the team used four major categories of characteristics: syntactic and semantic characteristics, automated support characteristics, methodology characteristics, and documentation characteristics. In examining syntactic and semantic characteristics, an important consideration was how close the ADL resembled Ada. Was it standard Ada, or was it a superset or a subset of Ada? If it were a superset of Ada, what language extensions were chosen? Regarding automated support, it was clear that such tools were essential for complex system development. Thus, the survey examined environmental support for each ADL. As we have already mentioned, the ADL should not stand in isolation, but should be one tool within a life cycle methodology. It was expected that the ADLs would support the design phase, but some ADLs offered support beyond this. Representation of this supplemental support information was an obvious point of interest for the study. Finally, documentation support was examined. They felt that it was necessary to have both tutorial and reference guides. Case studies, style guides, and other classes of documentation were also known to be important. Given this evaluation framework the sixteen ADLs were examined and compared in detail. We summarise the major findings below.

On a high level, the ADLs showed some convergence and agreement on issues of broad form. However, on a detailed level there was considerable divergence in implementation. Many of the differences were seen to be caused by open questions that lacked a generally accepted solution. Differences in syntax were often the result of different approaches to life cycle support. For example, all ADLs supported both preliminary and detailed design activities. However, while most ADLs extended the language to support other phases, there was little agreement on which kinds of supplemental information to represent. For example, some concentrated on interface specifications and others addressed requirements tracing and design constraints. There was considerable divergence in details.

The areas of automated support and methodology support were somewhat intertwined. There was a trend toward more automated tools, but few plans to automate such life cycle support activities as requirements tracking or design analysis. This seemed to be because many developers desired methodology independence for their ADL. The survey authors observed that methodological independence would have to be abandoned before comprehensive automated support tools could be developed.

The survey concluded that ADL efforts to this point concentrated mainly on syntax and semantics issues. A major unanswered question was how to fit the ADL into the larger context of the life cycle. The authors did not find any one implementation or group of implementations as clearly superior. Most had important weaknesses as well as strengths. They concluded that careful monitoring of new developments was important.

Since the 1984 survey, no one has produced a similar, comprehensive report. Today, the most appropriate place to obtain current ADL information is in the Association for Computing Machinery (ACM) publication "Ada Letters". Each month Ada Letters publishes an "ADL Developer's Matrix". The matrix is actually a table listing all known ADLs divided into two categories, ADL products that are "Commercially Available" and "Other". Entries in the "Other" category are usually ADLs that are for internal company use only, are products under development, or are government reports in the public domain. The information in the table consists of the organisation's point of contact, comparison of the ADL to Ada, available support tools, and product availability. In the November/December 1987 issue of "Ada Letters" there were fifteen commerically available ADLs and thirteen other ADLs (5). This represents nearly a doubling of ADLs from the 1985 NAC report.

With so many ADLs currently in use, it might be expected that some standardisation would have recently taken place. We address this effort in the next section.

4. IEEE Standardisation Attempt

In May 1982, the Institute for Electrical and Electronics Engineers (IEEE) formed a working group to examine the issue of possible ADL standardisation. Even after several years of work on ADLs within government and industry it became apparent to the group that the state of practice was still too immature to specify a standard. Instead the group published a "Recommended Practice" document in January 1987 (6). The rather brief nature of the document (about 3½ pages of main body) demonstrated just how far away ADL standards still were. The report is summarised below.

With regard to scope, it is instructive to see that the report explicitly does not specify: a single ADL syntax, a specific methodology, or the method of processing the ADL. What the report does specify is high level characteristics of an ADL with regard to methodology support, design support, supplemental support, and the relationship to Ada. Obviously, all these topics are covered only briefly.

For example, in the area of general methodology support, the report recommends support for the principles of abstraction, decomposition, information hiding, stepwise refinement and modularity, devoting only a few sentences to each topic.

One potentially controversial recommendation is that an ADL should conform syntactically and semantically to ANSI/MIL-STD Ada. Extensions to the would language consist of comments, possibly with special sentinel characters to alert any particular ADL preprocessor that might be used. This is possibly controversial because some believe ADLs should extend the language and require that the ADL be processed by a preprocessor before the ADL would become compilable Ada code. However, we believe the IEEE position on this point is correct if we are ever to converge on a standard.

Further summarisation here is unnecessary due to the brevity of the original document. What is most important about the report is that a highly qualified committee was unable to come close to defining a standard. This is a clear signal that general guidelines are very important for large governmental organisations, like NATO, who are trying to put some limits on the dialects of systems they will have to maintain.

5. ADL Guideline Documents

As was mentioned, many national policies dictate the use of Ada for certain applications. Even further, use of an ADL is required by some major organisations such as the US Army and Transport Canada. Other major organisations have required use of an ADL for specific projects. Given the lack of a credible standard and only high level convergence among ADLs, guidelines become essential for policy implementors. They are necessary to insure consistent and proper professional practices among contractors and to place at least rough bounds on the different ADLs to be maintained. Below we will discuss the major reports produced for Transport Canada and the US Naval Avionics Center. While we recognise the contribution of Ramtec to the US Army (7), we believe the other two reports are more generally applicable.

5.1 Transport Canada effort

One of the early efforts to define a set of guidelines was developed by PRIOR Data Sciences for Transport Canada, the Canadian federal transportation regulation agency. The original report was published in 1984 and was updated in 1987 (8). In this section we discuss the major features of the report.

The purpose of the report is to provide project managers a reference to assist in the evaluation of contractor proposals for new software systems. As mentioned, policy requires the use of an ADL within this agency. The document consists of three main sections: an overview of the PDL concept, an overview of the ADL concept, and ADL evaluation procedures. The appendices of the report include a complete design example, a discussion of design decision deferral, guidelines for implementations in languages other than Ada, and an ADL evaluation checklist.

One of the major contributions of this report is the identification of software development activities that can be supported by an ADL and how the ADL might support them. The guidelines identify these activities as requiring differing support from an ADL: design engineering, quality assurance, project management, independent verification, and customer acceptance. These classes of activities form the basis for choosing ADL features, which are then discussed on a high level.

Another important contribution is the appendix including a detailed design example for an organisational membership list management system. This demonstration of an actual ADL design includes almost 90 pages of ADL listing. The automated support tools utilized were an Ada compiler and an ADL tool. They discuss the features of the ADL tool which includes, among other things, several cross reference generators, a data dictionary, and a requirements tracer.

Perhaps the most widely recognised and popular section of the report is the ADL Evaluation Checklist. This checklist provides 18 pages of evaluation questions to apply to a potential ADL. It is divided into the following categories: support for the development activities, support for software engineering, actual features of the ADL, and tool support. The questions are phrased such as that a "yes" answer is desirable and provide a good basis to separate the good ADLs from the questionable.

In general, these guidelines enjoy a good reputation and have served as a model for many other efforts. They have been used successfully in several projects (9). However, as with all general guidelines, the success of any given ADL project is dependent on the skill of the people involved.

5.2 The Software Productivity Solutions Guidelines

In 1985 the US Naval Avionics Center once again took a leadership role in the ADL arena by letting a contract for the production of ADL guidelines to be compatible with the US DOD-STD-2167. This standard is the US DoD standard for development of defense software systems. In January 1987 the contractor, Software Productivity Solutions (SPS), issued their report (10). The main body of the report tackles the ADL question by a series of propositions and resolutions identifying the major issues to define an ADL's scope. Included as major appendices are ADL development guidelines, acquisition management guidelines, and guidelines for tool sets. Generally, the report is compatible with the Transport Canada Guidelines. This report was able to extend and formalise some of the recommendations and discussions initiated in the Canadian report. Below we discuss the major features and contributions.

The main body of the report addresses propositions and states resolutions about the scope of an ADL in the following main categories: support for engineering activities, support for management activities, support for quality assurance activities, automated support for the ADL, and the relationship between Ada and an ADL. The verification and acceptance activities used in the Canadian report are generally subsumed among the categories here.

The ADL development guidelines appendix are more detailed than any guidelines produced thus far. They focus in particular on the support requirements for engineering design. Using these as a focus, they recommend specific ADL syntax and features using an approach that specifies both kernel and optional ADL features. Kernel features are those that should be supported by all ADLs with options chosen to meet the special needs of a given project. The main contribution to previously suggested ADL syntax was support for the DoD-2167 "static structures" model for software systems. The static structure of a system is a way of specifying an arbitrarily complex hierarchy of software. Given the size of the DoD market, many international vendors produce products that are compatible with the standard.

While the acquisition management guidelines are particularly useful to those using a US DoD system acquisition model, they can be tailored to other models. In the guidance on procurement, there are suggestions for the request for proposals and proposal evaluation. In the guidance for contract monitoring there are suggestions for technical reviews, verification and audits.

While the report has initially attracted much interest, there is some controversy on the mapping of Ada constructs to the 2167 model. This is not necessarily a criticism of these guidelines but rather concern by Ada designers of mapping to the static structure model itself. Those interested in more details are referred to (11) and (12).

6. Conclusions

In this paper we have discussed some of the more important work evaluating the state of ADL practice today. We have seen that there is high level convergence of philosophy on many issues, but there are many open questions on the detailed level. This has resulted in abandoning attempts to standardise and concentrating on providing *general guidelines. Most people seem* to agree that guidelines for general usage must recognise the unique requirements of each project and choose the ADL features accordingly. This points in the direction of the "kernel with options" philosophy of the SPS report.

A potentially fruitful area for further work is the definition of guidelines for what could be a set of "standard" options. This is alluded to in the SPS report, but not developed in detail. For instance, some specific projects might have a need for an ADL that would incorporate database aspects. Since not all projects would require this, database specific ADL features should be options and not part of the kernel. However, the requirement to develop systems with database aspects would occur often enough to call for common guidelines for this application. If you extend this concept to other common requirements, you could develop a standard set of ADL feature "modules". With enough refinement and use these could provide the basis for a new standardisation effort.

Unquestionably the major area for future work is in the area of methodologies. This problem is not unique to Ada and ADLs, but some of the opportunities presented by the language are not fully integrated into a generally accepted life cycle approach. Many widely recognised Ada development methodologies have limitations or rely on above average software engineering skills for their effective use. In order to promote good system development by the average software engineer, we need simple, well defined methodologies backed up by substantial automated support. As methodologies evolve, the role of ADL as a tool may also evolve, but it seems apparent that ADLs will be a major tool for Ada system development for the foreseeable future.

REFERENCES

1. Booch, G., Software Engineering with Ada, Second Edition, Benjamin/Cummings, Menlo Park, CA, 1987.

2. Caine, S. and Gordon, E., "PDL - A Tool for Software Design", Proceedings of the 1975 National Computer Conference, American Federation of Information Processing Societies (AFIPS), 1975.

3. SofTech, Ada Programming Design Language Survey Final Report, Report for Naval Avionics Center, Contract N00163-82-C-0030, Naval Avionics Center, Indianapolis, IN, October 1982.

4. Computer Technology Associates, Survey of Ada-Based PDLs, Report for Naval Avionics Center, Contract N00163-84-C-0300, Naval Avionics Center, Indianapolis, IN, January, 1985.

5. Kerner, J., "Ada Design Language Developers 9/11/87", Ada Letters, Volume vii, Number 7, November/December, 1987.

6. Institute of Electrical and Electronics Engineers (IEEE), IEEE Recommended Practice for Ada as a Program Design Language, IEEE Std 990-1987, January, 1987.

7. Ramtec, An Interim Guideline for Ada Based Development and Product Design Documentation, Report for US Army Communications Electronics Command, Contract DAAK20-83-G-0711/BG-4B, US Army CECOM, Fort Monmouth, NJ, 31 October 1984.

8. PRIOR Data Services, Ada Program Design Language (PDL) Evaluation Guidelines, Report for Transport Canada, Nepean, Ontario, January 1985.

9. Leavitt, P., "Ada PDL Evaluation Guidelines", NATO Research Study Group Workshop on Distributed System Design Methodology, Brussels, Belgium , 15-18 September 1987.

10. Software Productivity Solutions, ADL Guidelines Final Technical Report, Report for Naval Avionics Center, Contract N00163-85Q1322, Naval Avionics Center, Indianapolis, IN, January 1987.

11. Grau, J, and Gilroy, K., "Compliant Mappings of Ada programs to the DOD-STD-2167 Static Structure", Ada Letters, Volume vii, Number 2, Mar/Apr 1987.

12. Software Productivity Solutions, "Proceedings of the Workshop on Ada-Based Design Languages", Melbourne, FL, February 1988.

DEBUGGING DISTRIBUTED ADA AVIONICS SOFTWARE

1Lt Marc J. Pitarys
United States Air Force
AFWAL/AAAF-3
Wright-Patterson AFB, OH 45433-6543
United States of America

SUMMARY

Future avionics systems will consist of distributed fault-tolerant architectures. The operational flight software will be written in the Ada programming language and use a distributed operating system. Developing and maintaining this software requires new and innovative debugging techniques to reduce cost, time, and complexity. This paper will describe two specific techniques. These are dynamic trace buffers and software Built-In-Support-Functions (*BISFs*). First the need for new approaches to debugging distributed Ada software is addressed. Then the implementation of each of the techniques is presented. The Avionics Laboratory is the only organization to successfully demonstrate both techniques with MIL-STD-1750A computers. Observations and recommendations for using these techniques will be reported.

PREFACE

There are two new advances in technology that will influence the development of avionics software. One is the introduction of the Ada programming language. The Department of Defense (DoD) has mandated the Ada programming language for all mission critical software. Since Ada contains many advanced features not found in other languages used in avionics systems, their impact on avionics software debugging and integration needs to be known. The second advance is the use of distributed fault tolerant avionics architectures. The new avionics systems will place additional demands on the software developer, integrator, and maintainer. To support these advances in technology, new software debugging approaches are required. The use of Built-In-Support Functions and trace buffers are two such approaches.

1. ADA TASKING

With the introduction of Ada, several new features are available to the programmer. For example, the *high-level construct tasking can be used for concurrent programming.* In the past concurrent programming relied on low-level constructs like semaphores. Semaphores will still be used and can be implemented with Ada. Understanding the implications of debugging software that uses tasking requires knowing something about how Ada tasking works.

Communication and synchronization between tasks are handled by entry calls and accept statements. When an entry has been called and a corresponding accept statement has been reached, the sequence of statements, if any, of the accept statement is executed by the called task (while the calling task remains suspended). This interaction is called a rendezvous. Thereafter, *calling task* and the task *owning* the entry continue their execution in parallel.[1]

It's important to note that if several tasks call the same entry before a corresponding accept statement is reached, the calls are queued. There is one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in order of arrival.[1] Debugging Ada tasks requires knowledge of when the rendezvouses occur and in what order. Therefore, the indeterminacy of Ada tasking requires monitoring techniques that can support nonprobabilistic code.

The delay statement is frequently used with Ada tasking. The execution of a delay statement suspends further execution of the task that calls the statement for at least the duration specified. A delay statement with a negative value is equivalent to a delay with a zero value.[2] Since the delay is minimal, errors could occur if software is relying on precise timing.

2. DYNAMIC ALLOCATION OF VARIABLES

The dynamic allocation of Ada variables also presents new challenges for debugging. Local variable addresses are determined by adding an offset to the stack. Thus, the actual addresses are unknown prior to execution. This complicates the monitoring of variables in real-time, especially in a fault-tolerant and reconfigurable environments.

Current software test equipment can support statically allocated variables. Ada's variables can be monitored if they are declared globally. However, if the Ada software is dynamically allocated to different address states, then even these variables cannot be monitored. Obviously the capability to monitor Ada variables is needed.

3. SOFTWARE FAULT-TOLERANCY

Software fault-tolerancy also requires new debugging approaches for development and integration. Future avionics systems will have some type of software fault-tolerancy. For example, the PAVE PILLAR specification calls for the use of dynamic error handling techniques. These techniques involve the use of operational software code to detect, confine, and correct software data and timing errors during software operation.[3]

Verifying that fault-tolerant software works correctly is a complex effort. The software test equipment will need to trace the execution of the avionics program in real-time. With a distributed system the program tracing must take place for multiple programs running on multiple processors. With the software running at Very High Speed Integrated Circuit (VHSIC) speeds this is beyond the capability of current debugging equipment. VHSIC also presents a physical signal access problem.

4. BISF

Current real-time software monitoring technology does not support dynamic memory/processor allocation. As mentioned above, future avionics systems will have these characteristics. The BISF concept was proposed as a solution to this problem by Duane O. Hague Jr., an engineer at the United States Air Force Wright Aeronautical Laboratories' Avionics Laboratory. Mr. Hague proposed the creation of a special class of software hook instructions that are embedded into the application and/or operating system software. The BISF instructions would transfer software hook information to a specialized BISF integrated circuit resident on the processor module that functions as an output port to the external support equipment. To the processor/system, execution of a BISF instruction would be equivalent to the execution of a "NO-OP" instruction.

There are three BISF instructions that would provide the needed debugging support for multiprocessor/Ada avionics architectures. These instructions are the External Flag (XFLG), External Flag Register Dump (XFLGR), and the External Trace (XTRC).

The XFLG instruction outputs an unique flag value to the specialized BISF output port, thus making the value available to the external support equipment. This provides the capability for tracing code execution. By time tagging the flag output software performance can be measured and timing and intermittent errors can be located.

The XFLGR instruction outputs an unique flag value and the contents of a register to the BISF output port. This provides the capability to monitor stack based variables. Software test equipment can use the value of the stack pointer to add with the compiler generated relative address of the program variable to determine the absolute address for monitoring.

The XTRC instruction triggers the test equipment to take a limited size snap-shot of sequential processor internal bus activity such as instruction or data reads and/or writes.

BISF instructions are inserted at points of interest in the Ada software. For example in Ada tasks, procedures, and exception handlers. Other areas consist of those sections of software that need performance tuning. XFLG instructions can be inserted by the compiler as part of the calling convention used for context changes. Using the BISF instructions requires no change to Ada (MIL-STD-1815A) or MIL-STD-1750A.

5. BISF IMPLEMENTATION

The System Evaluation Branch of the Avionics Laboratory initiated an in-house effort in 1986 for full BISF proof-of-concept for the MIL-STD-1750A computer. The instructions were implemented by using the optional user defined Built-In-Function (BIF) instruction in the MIL-STD-1750A Instruction Set Architecture. The BIF instruction invokes special operations defined by the user.[3] The XFLG and XFLGR instructions were demonstrated with Ada running on a Fairchild F9450 1750A computer in the fall of 1987. Currently a software Performance Monitor and Controller (PMAC) that will use the BISF for monitoring and debugging distributed Ada software running on 1750A computers is being designed.

The F9450 1750A cpu was selected because it supports the BIF instruction. The F9450 implements the BIF as a three word instruction. It treats the instruction as a co-processor escape where the BIF instruction writes its extension field to a dedicated XIO address. The implementation of each of the BISF instructions for the F9450 is shown in figure 1.

There are two approaches to using BISF with Ada software. One approach requires the Ada compiler having capability for inserting machine code inline, and the assembler allowing for the BIF instruction or the capability for defining an instruction. This approach requires pragma in-line and pragma interface for including machine code with the Ada software. The other approach involve the use of the package MACHINE_CODE. An example of each approach is shown in figures 2 and 3.

Several experiments using the BISF with Ada were conducted. One experiment involved using the XFLG instruction for monitoring a program similar to what is shown in figure 2. Each Ada task was to perform an activity at a regular interval. To verify that indeed the tasks were operating within the given time constraints the XFLG instruction was used. Using a logic analyzer to monitor the BISF port, the time between each identical flag was obtained. Each task was verified to execute within its allotted amount as specified in the program.

6. DISTRIBUTED OPERATING SYSTEMS

Application software written for next generation avionics systems will use an embedded real-time distributed operating system. This operating system provides the control mechanisms and application services within the multiprocessor system. The processes that make up the application software are scheduled by the operating system of the avionics system.

Programming a distributed architecture is not a trivial activity. The problems of race conditions, deadlock, and process starvation occur during the development of the avionics software. To assist the programmer the distributed operating system can employ trace buffers.

7. TRACE BUFFERS

An Ada distributed Kernel Operating System (KOS) was developed and delivered to the Avionics Laboratory by Westinghouse Defense and Electronics Center for the VHSIC Avionic Modular Processor (VAMP) architecture. The KOS is currently being used with the Engineering Development Model (EDM) of the VAMP. The EDM is a multiprocessor system consisting of three 1750A cpus, a 1553B board, and a bulk memory card connected to a parallel internal (PI) bus.

To assist in debugging of the application software the KOS employs two trace buffers - an Active Trace Buffer and a Call Trace Buffer. The buffers are optional, enabled by setting a parameter in the KOS's definition file. Both buffers are circular with their size specified in the definition file.

The Active Trace Buffer is a chronology of the active process in the processor.[4] The active process information is entered into the buffer by the KOS. This information is hex data that indicates what KOS services were called.

Usually information is needed on all the processes that were active in the system. The call trace buffer is comprised of 4-word entries, where each entry contains information about calls made by the application to the KOS or when PI-BUS messages are received telling the KOS to signal a semaphore/event.[4] A sample dump and explanation is given in figure 4.

From experiences in the laboratory, the trace buffers reduce debugging time. They are extremely helpful in determining if the system is correctly initialized. The programmer can verify that all processes are defined and the software starts its execution correctly. Time is also saved by avoiding the work of going through listings and link maps to determine what process was the last one active.

8. CONCLUSION

Debugging distributed Ada avionics software can be a complicated job. The BISF is the lowest cost, least complex approach to real-time software debugging and monitoring. The Avionics Laboratory has shown that BISF instructions can be implemented with Ada and the MIL-STD-1750A instruction set architecture. Using trace buffers is another low cost approach. Their usefulness has been demonstrated with a distributed operating system running on an architecture similar to what will be used by future avionics systems.

9. REFERENCES

1. ANSI/MIL-STD-1815A Ada Programming Language; Philadelphia, PA; Naval Publications and Forms Center; 22 JAN 1983; p9-9

2. ANSI/MIL-STD-1815A Ada Programming Language; p9-10

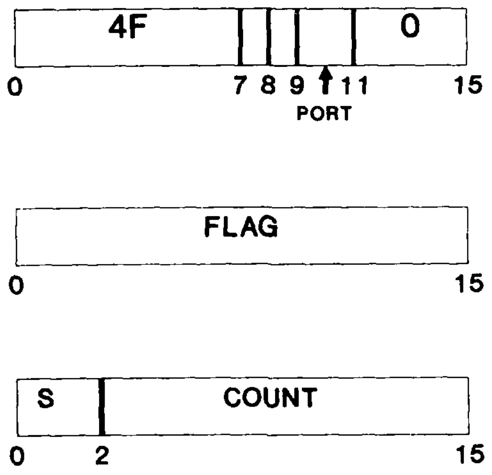
3. Architecture Specification for PAVE PILLAR AVIONICS; WPAFB, OH; Avionics Laboratory; JAN 1987; p15

4. MIL-STD-1750A Notice 1; Washington, DC; Government Printing Office; 21 MAY 1982; p138a

5. VHSIC Avionics Modular Processor Kernel Operating System User's Manual; Baltimore, MD; Westinghouse Defense & Electronics Center; 19 NOV 1987; pC-1

XTRC

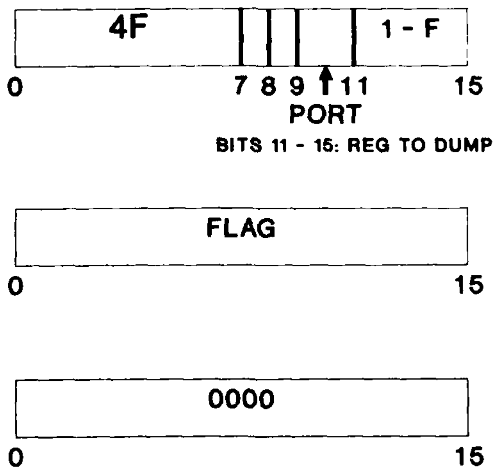
BIT 8 CONTAINS 0 BIT 9 CONTAINS 1



S = TRACE SELECT

XFLGR

BIT 8 CONTAINS 0 BIT 9 CONTAINS 1



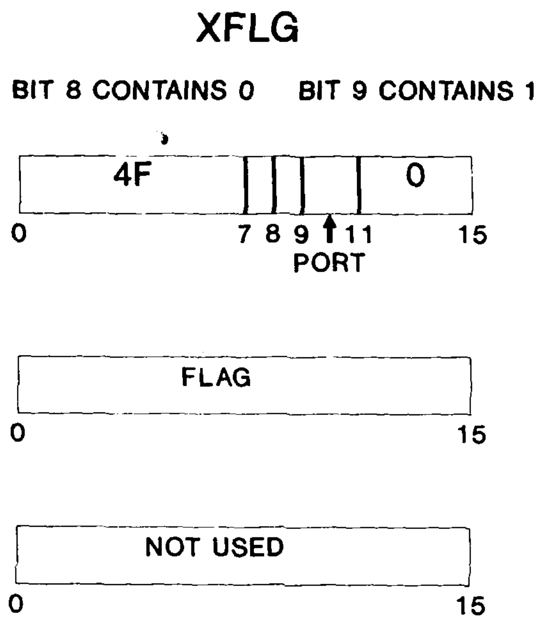


FIGURE 1

```
with SYSTEM; use  SYSTEM; package BISF_SUPPORT is

     procedure XFLG0;
     pragma interface(ASSEMBLER, XFLG0);
     pragma inline;

     procedure XFLG1;
     pragma interface(ASSEMBLER, XFLG1);
     pragma inline;

     procedure XFLG2;
     pragma interface(ASSEMBLER, XFLG2);
     pragma inline;

                  .
                  .
                  .
end BISF_SUPPORT;
```

This package is included with the Ada applications software

```
with BISF_SUPPORT;
use BISF_SUPPORT;
with SYSTEM;
use SYSTEM;
with CALENDAR;
use CALENDAR;
procedure AVIONICS_TASKS
```

```
     task type TASK0_TYPE is
          entry START;
     end TASK0_TYPE;

     task type TASK1_TYPE is
          entry START;
     end TASK1_TYPE;

           .
           .
     additional task type declarations
           .
           .

     TASK0 : TASK0_TYPE;
     TASK1 : TASK1_TYPE;

           .
           .
     additional task declarations
           .
           .

     task body TASK0_TYPE is

          NEXT_ITERATION_TIME : CALENDAR.TIME;
          CYCLE_DURATION      : constant := 0.004;

     begin
          accept START do
              -- MAIN PROGRAM IS NOW WAITING FOR RENDEZVOUS COMPLETION
               null;
          end START;

          NEXT_ITERATION_TIME := CLOCK;

          loop
--BISF                  XFLG0;
          --Perform avionics activity
                        .
                        .
                        .
               NEXT_ITERATION_TIME := NEXT_ITERATION_TIME + CYCLE_DURATION;
               delay (NEXT_ITERATION_TIME - CLOCK);
           end loop;

      end TASK0_TYPE;

     task body TASK1_TYPE is
          NEXT_ITERATION_TIME : CALENDAR.TIME;
          CYCLE_DURATION      : constant := 0.008;
          -- Activity performed once every 8 msec.

     begin
          accept START do
           -- MAIN PROGRAM IS NOW WAITING FOR RENDEZVOUS COMPLETION
              null;
          end START;

          NEXT_ITERATION_TIME := CLOCK;

          loop
--BISF                      XFLG1;
               --Perform avionics activity
                            .
                            .
                            .
               NEXT_ITERATION_TIME := NEXT_ITERATION_TIME + CYCLE_DURATION;
               delay (NEXT_ITERATION_TIME - CLOCK);
           end loop;

      end TASK1_TYPE;
```

```
                        .
                Additional Task bodies
                        .
                        .

begin       --of TASK_EXAMPLE
            --ALL TASKS ARE SUSPENDED AT ACCEPT START, AND

    TASK0.START;
    TASK1.START;

        .
        .
     Additional task starts
        .
        .

end AVIONICS_TASKS;
```

Figure 2

```
with System; use System;
package Machine_Code is
    BITS : constant := 1;
    WORD : constant := 16;
    type Opcode is (BIF);
    for Opcode'SIZE use 8*BITS;
    for OPCODE use (16#4F#);
    type MANDATORY_TYPE is range 0..3;
    for MANDATORY_TYPE'SIZE use 2*BITS;
    type PORT_TYPE is range 0..3;
    for PORT_TYPE'SIZE  use 3*BITS;
    type REGISTER_TYPE is range 0..15;
    for REGISTER_TYPE'SIZE use 4*BITS;
    type FLAG_TYPE is range 0..65535;
    for FLAG_TYPE'SIZE  use 16*BITS;
    type BISF is
        record
          INSTRUCTION    : OPCODE;
          MANDATORY      : MANDATORY_TYPE;
          BISF_PORT      : PORT_TYPE;
          REGISTER       : REGISTER_TYPE;
          FLAG           : FLAG_TYPE;
          NOT_USED       : FLAG_TYPE;
        end record;
     for BISF use
        record
          INSTRUCTION at 0*WORD range 0..7;
          MANDATORY   at 0*WORD range 8..9;
          BISF_PORT   at 0*WORD range 10..11;
          REGISTER    at 0*WORD range 12..15;
          FLAG        at 1*WORD range 0..15;
          NOT_USED    at 2*WORD range 0..15;
        end record;
end MACHINE_CODE;
```

A BISF instruction can now inserted into the Ada code by including the MACHINE_CODE package and using a statement similar to what's given below:

```
BISF'(BIF, 1,  0, 0, 63, 0);
```

Figure 3

Trace Buffer Dump	Translation
00FE0006FFFF0001	Process 254 defines Process 1
00FE0006FFFF0002	Process 254 defines Process 2
00FE0006FFFF0003	Process 254 defines Process 3
00FE0006FFFF0008	Process 254 defines Process 8
00FE0006FFFF0009	Process 254 defines Process 9
00FE00020047FFFF	Process 254 waits on Semaphore 47
000900020013FFFF	Process 9 waits on Semaphore 13

Figure 4

DISCUSSION

M.Muenier, Fr

I can see a potential problem in instrumenting the code for debugging purposes and the possible aiding of synchronisation problems that could cause deadlocks. Could you comment on this?

Author's Reply

The BISF instructions are single machine code instructions that remain with the software after it is fielded. They consume less than 1% of the memory and CPU thrust. The instructions act like no-op instructions and therefore do not change the state of the system. The hook information is transferred directly to the output post.

C.Berggren, US

Your system diagram shows a separate BISF bus. Would you please tell us what is it and how you intend to use it?

Author's Reply

The BISF bus transfers the data output from the BISF instructions to external test equipment or an inflight monitor. Test equipment only needs to monitor this bus to make use of the BISF.

AUTOMATED ADA CODE GENERATION FOR MILITARY AVIONICS

Robert S. Ardrey, II
Computer Technology Associates, Inc.
900 Heritage Drive
Ridgecrest, California 93555
USA

SUMMARY

This paper presents our experience with Ada-directed software development methodologies. These include functional programming the use of class instances object-oriented design, and Ada as a program design language. The paper builds upon prior experience by describing an automated Ada code generation environment. The environment addresses the enhancement of existing software tools, new development currently underway, and compiler support for military flight computers such the AN/AYK-14. Finally, military avionics applications of such an environment are discussed. Predicted improvements in the areas of prototyping, productivity, reusability, and maintainability are examined.

1. INTRODUCTION AND BACKGROUND

The United States Department of Defense Ada language initiative has given rise to a sense of mystique within the weapon system community. This is largely a side effect that results from confusing Ada linguistics with development methodology. As the demand for mission critical avionics software continues to accelerate, it is necessary that our industry demystify the use of Ada and bring effective software engineering techniques to bear.

This paper suggests the principle focus of avionic software development should be on design, and that production of Ada (or other) source code should be automated.

1.1 The Ada Imperative

While debate continues concerning the eventual acceptance of the Ada programming language, there is increasing evidence of commitment to Ada on the part of the Department of Defense. Last year, Salomon Brothers published a thorough review [1] of Ada's status. Selected details of the report are summarized below.

In 1975, the High Order Language Working Group performed two significant actions. First, they officially limited the languages that could be used on defense contracts. Second, they established a set of requirements for a new language that could be used for all embedded computer systems. The final language design was selected in 1979, and was called Ada.

In 1983, Ada was recognized as a standard by the Department of Defense, and by the American National Standards Institute. Ada has also recently been accepted as an international standard. In 1983, Ada was stipulated for use in all new mission-critical software development programs.

In spite of this stipulation, many waivers were requested and granted in the 1983 to 1986 time frame. Largely due to management concerns about effects on schedule, cost, and reliability, this situation resulted in only limited acceptance of Ada. In 1987, two new Department of Defense Directives (3405.1 and 3405.2) were issued. These directives jointly emphasized the requirement to use Ada on defense contracts, and made the waiver process much more difficult.

The commitment to use Ada on several major, current system acquisitions (such as Space Defense Initiative, Advanced Tactical Fighter, and Federal Aviation Administration Advanced Automation System), suggest that the Ada imperative is real. Therefore, it is in the best interest of the military avionics industry to consider the near-term use of Ada as likely, and to evaluate the implications of this eventuality.

1.2 Trends In Avionics Software Criticality

The criticality of military avionics and associated software is increasing dramatically. This is but one of the conclusions supported by the "Military Avionics in the 1990's" seminar series offered by Technology Training Corporation. Information relevant to this conclusion is discussed in this section.

The percentage of military aircraft acquisition cost attributable to avionics has effectively doubled in the last twenty years. This increase is in spite of improved hardware performance and decreased hardware cost. The percentage growth in cost of avionics is the result of a marked increase in the amount of embedded avionic software.

The increase in avionic software during this period has been as much as a full order of magnitude. There are three important reasons for this increase. First, there has been an increase in the overall functionality of the weapon systems. The quantity of software required to perform the additional functions has kept pace. Second, there has been a somewhat surprising shift in the functional partitioning between hardware and software. Software has grown from perhaps ten percent of the total functionality to as much as thirty-five percent. Thus, even in the absence of other factors, the amount of software would increase. And third, the growing pilot workload is now at, and in some cases well beyond, the point of saturation. To counter this requires a significant redefinition of the Pilot-Vehicle Interface, with a corresponding increase in the volume of software.

The increasing amount and complexity of this embedded avionic software would constitute a development challenge even if the required reliability were constant. However, this is not the case. An additional factor is the increasing criticality of modern avionics. There has been a transition that started with mission importance (digital displays, route planning, stores inventory), has passed through mission criticality (terrain following, weapon control, inertial navigation), and has now progressed to flight criticality (unstable flight, fly-by-wire, engine control). As the role of the avionics becomes more critical, the required software reliability increases.

As evidenced by the Advanced Tactical Fighter program, the trend in avionics continues toward increasing amounts of software and increasing reliability requirements. Unfortunately, the United States STARS (Software Technology for Adaptable Reuseable Systems) Foundation estimates a programming personnel shortfall on the order of four percent per year. Highlights of an approach to reconciling these issues, while addressing the Ada imperative, are presented in the next section.

1.3 Automatic Programming in Ada

Estimates of the length of time required to adequately train an "Ada developer" are on the order of six to twelve months. This is certainly enough to suggest that Ada might be very difficult to use. In point of fact, however, an experienced programmer can learn Ada syntax and semantics in one week, and be generating functional code in two weeks. Given some experience with more modern high-order languages, such as Pascal or MODULA-2, one can be writing Ada code of reasonable quality within one month.

There is an apparent contradiction here. The reason is that, unlike most of its predecessors, the Ada language was designed to provide integrated support for modern programming practices. In addition, the Ada initiative emphasizes state-of-the-art software engineering. The training estimates, therefore, actually consist of two parts: becoming a qualified software engineer, and learning how to program in Ada. How then should industry approach the training of Ada developers?

The business community provides a model that applies to this situation. Problem analysis and specification have been separated from the implementation process. Fourth generation languages address the problem domain, while application generators offer an automated approach to the solution space. Training emphasis is on problem analysis; the use of non-procedural languages enhances productivity; and the reuseable application generators are more cost-effective than manually creating problem-specific code.

Consideration of the above model, in the context of Ada and military avionics software, suggests the following approach:

- concentrate on software engineering training
- support design methodology with automated tools
- incorporate mechanisms for design and code reuse
- automate the generation of Ada source code
- focus on military avionics and processors

Section 2 discusses techniques that constitute a foundation for automatic programming in Ada.

2. RELEVANT METHODOLOGICAL EXPERIENCE

Computer Technology Associates, Inc. (CTA) is applying modern software engineering methodology to the process of Ada code development. The work is being performed through government contracts, the Small Business Innovative Research program, and Independent Research and Development tasks. This section discusses several activities that are relevant to Ada code generation.

2.1 Ada As a Program Design Language

Perhaps one of the earliest popular methodological efforts was the use of Ada as a program design language. CTA, like many other organizations, attempted to use Ada in this way. Early efforts within the industry were hampered by a widespread lack of uniformity in approach. Although a recent standard [2] has been issued that provides a set of recommended capabilities, it does not address the actual usage of the design language.

CTA's experience with Ada as a program design language was useful for intercomponent interface specification (via the Ada package mechanism), data structures (via Ada declarations), and control representation (via logic constructs and procedure invocation). However, there were two important areas that did not provide any benefits. The first was the effort required to specify a design in the rather verbose Ada syntax; and the second was the need for formal design methodologies.

2.2 Object-Oriented Design

In his paper on object-oriented development, Booch [3] has suggested that classical, "structured" development methodologies do not lend themselves well to Ada language features. This is due to a function-centered view that does not fully utilize such Ada features as abstraction, genericity, and encapsulation. He makes a case for a data-centered approach that extends the system development concepts developed by Jackson [4].

CTA has found this object-oriented methodology to be a sound one, and well suited to coding in Ada. The close parallel between problem domain elements and objects in the solution space produces a design that is correct by inspection. This is at odds with the usual arguments of correctness that are based upon the mapping from structured designs back to the problem domain.

CTA has been developing software tools for object-oriented design development efforts. There are now several tools that support a range of activities, from requirements specification (using both data and functional views) through design (with object diagrams, editors, and validity checks). Experience on several small Ada development projects has demonstrated the effectiveness of both the approach and the tool set.

2.3 Automatic Programming Techniques

There are two areas of research that CTA has been pursuing with respect to automatic programming. The first, referred to as Ada prespecifications, considers automated code production concepts such as Martin [5] describes for fourth-generation languages. The underlying technique is a generative one, in which a tailored language is applied to a restricted problem domain; the resulting prespecification is used to automatically generate source code that can be compiled for subsequent execution. CTA is using the Ada language itself [6], to avoid the proliferation of languages that would otherwise result from an application that spanned multiple problem domains. The Ada prespecification includes declarative structures in place of a domain-specific metalanguage, and a library of code body generators. Compilation and execution of the prespecification results in an Ada code body that implements the specified design. A code expansion ratio of 17-to-1 for Ada source lines from prespecification lines has been observed.

The second area, referred to as functional programming, is derived from the use of DeMarco's [7] data flow diagrams as a means of specification. This is a constructive technique, where a set of primitive functions are defined from which the diagram can be composed. The data flows that connect the functions constitute function parameters and results. Constraints are used to ensure that the diagram construction is valid. The resulting symbolic representation is the input to a translation process that allows direct execution (interpretively) of the specification. CTA has demonstrated this technique in a system that provided for graphic construction of a series of algebraic computations.

2.4 Classes and Inheritance

While Ada was not specifically designed to be an object-oriented programming language, it does have features that support the general approach. There are some important concepts, however, such as the class hierarchy and inheritance capabilities of the object-oriented Smalltalk language [8], that are not well supported by Ada. Seidewitz [9] gives examples of these concepts, and shows how they might be represented in Ada programs. Unfortunately, it is necessary for the programmer to manually elaborate the design with applicable Ada constructs. This is not an optimal approach, from either the development or maintenance point of view.

CTA has developed an alternate mechanism for incorporating classes and inheritance. Using the Ada prespecification method discussed above in section 2.3, class definitions, hierarchies, and inheritance information are provided in the prespecification declarations in a form designated as a "phylum". An invocation is made upon the phylum Ada code generation function, with the associated code body library. The result (after compilation and execution of the prespecification) is a compliment of Ada packages, interfaces, and procedures that provide the necessary structure for the design implementation.

2.5 Reuse Concepts

CTA has evaluated concepts for the reuse of Ada development products. Perhaps the most obvious approach is reuse of the code itself. For example, Booch [10] has defined a taxonomy for data structures and manipulation, and has developed a collection of well-documented, validated Ada software components. This concept does not address tailored implementation that results from the target environment. Neither does this concept lend itself well to application-specific problems, where requirements differences may necessitate alternate designs. The notion of a taxonomy is an important aspect of this approach, as it provides a method to categorize and organize the code bodies. While classification is not discussed any further, it is central to all reuse concepts.

A second approach is the reuse of design components. These "objects" encapsulate data structures and a set of operations on the data. This allows reuse at a higher level of abstraction, in a manner that is closer to the problem domain. Tailoring for the target environment is accomplished during implementation. If an automatic code generation mechanism is used, object reuse can be an effective labor reduction technique.

A still higher level of abstraction is available through the use of classes, as discussed in section 2.4, and the inheritance mechanism. This "specification reuse" concept is inherent in the inheritance process, with application-specific adaptation through overriding selected superclass capabilities, or by the addition of new capabilities to the subclass. Bailin [11] presents CTA progress towards a software reuse environment that is based upon the class hierarchy approach.

3. AN AUTOMATED ADA CODE GENERATION ENVIRONMENT

This section proposes environmental concepts that support the automated generation of Ada source code. The concepts are discussed presented within the context of a classical system development life cycle, as listed below.

- system requirements specification
- system architecture definition
- element requirements analysis
- element design synthesis
- element implementation
- integration, test, and evaluation
- operation and maintenance

Note, that while an architectural element consists of hardware and/or software, this paper addresses only the software portion. Issues concerning "software first", "software in hardware", and "software/hardware partitioning" are not discussed.

The focus of this section is on software: specification of requirements and design, code generation, and support for target avionics processors. Application of object-oriented principles to system requirements and architecture appears to be premature at the present time, although capability-based computer systems are likely to change this situation.

3.1 Software Requirements and Design Specification

Historically, the processes of requirements analysis and design synthesis have been differentiated by the characteristics of their respective engineering activities. It could be argued that this is an artificial distinction, resulting from the problem domain and solution space dichotomy that section 2.2 discussed. CTA has found that object-oriented development blurs this distinction. The analysis process defines hierarchies of objects, each with data and function content. The lowest level of the object "decomposition" becomes pure function. Synthesis, then, is the process of adding design detail, rather than one of recasting the problem domain into a solution space.

The foundation of the development environment, therefore, is an object-oriented development approach. This includes the class and inheritance concepts of section 2.4, where the class phylum consists of problem domain objects. This does not suggest that there is no role for traditional structured analysis techniques. CTA's recognition of this continuing role has been to develop a tool by which, given an both "annotated" data flow diagram and an entity relationship diagram, a preliminary object hierarchy is generated. The object-oriented design is then completed through the use of graphic editors. While additional development would be required to accommodate other possible forms of inputs, the framework has already been developed and is in use at CTA.

The issue of formal specification is more complicated. The Department of Defense stipulates a the development life cycle, standards to be employed, and documents to be produced. These are not well suited to object-oriented development and resulting specifications. A number of vendors are beginning to address the semiautomatic generation of standard specifications from internal requirements and design representations. A partial list of such products and vendors is given below.

- STATEMATE, by i-Logix
- CARDtools, by Ready Systems
- TekCASE, by Tektronix
- TAGS CASE2, by Teledyne Brown
- superCASE, by Advanced Technology

Typically, the tools are based upon structured analysis, rather than object-oriented development. CTA has not yet assessed the practicality of using their internal data representations as inputs to the object generator. A recent software development standard [12] does suggest a more liberal approach to the life cycle, that could encompass object-oriented development. The required specifications do not offer such latitude, however. The conclusion, therefore, is that evolution of the mandated process and products must continue in order for object-oriented software development to be accommodated.

3.2 Automated Source Code Generation

For the purpose of this paper, automated generation refers to any mechanism from which source code results other than manual entry. For example, perhaps the most obvious "automated" technique is reuse of existing code. While this is actually a constructive approach, rather than a generative one, it does not require manual code development. Automated support is provided by a reuse environment that facilitates identifying appropriate code bodies as a consequence of the object-oriented requirements and design elaboration process. To increase the probability that the appropriate code bodies exist and can be identified, it is important that they be originally developed through the *object-oriented facilities of the reuse environment*.

A second automated approach is that of Ada prespecification. This is a generative technique, that creates Ada code from the problem domain metalanguage. In general, any system object could be a candidate for having its own metalanguage. The functional aspects of the object would be expressed in a prespecification, with subsequent generation of the Ada code. CTA is currently in the process of incorporating this capability into the existing object-oriented development tool set.

Metalanguage prespecifications will not be applicable to all types of objects. This is especially true for computationally intensive numerical applications. There are other techniques to support code automation in these cases. One is the use of Ada as a program design language. Algorithmic details are expressed in Ada, with the type and variable bindings established through an automated generation step. CTA is also incorporating functional programming as a design elaboration technique. Specifically, a set of functional primitives (code fragments) can be associated with data elements through the data flow diagram. An automated construction step will bind the Ada code and data for inclusion in the final packages.

In all of the above techniques there will remain a need for manual editing, tailoring, and selective optimization. This can be provided through syntax-directed editors, such as currently available in CTA's object-oriented development tool set. Note that manual modification introduces a new variant of a code body. While its position within the object phylum is unchanged, it becomes available only through construction; generation could no longer be used to produce the variant automatically.

3.3 Target Processor Support

The Ada language provides a number of mechanisms to match a program to its target processor. These include representation specifications, pragmas, and implementation dependent features. The use of these mechanisms is a manual, labor-intensive activity for which automated support would be very beneficial. Further, the inclusion of such information affects the reusability of the software. One approach would be to include the target processor specifics as library variants, thus allowing their inclusion in a transparent manner. This approach has the disadvantage of many versions of library modules being required. There is evidence in the industry that an alternative approach is gaining emphasis, that of rule-based tailoring. The benefit of this approach would be the retention of only general forms of the software, with the target processor specifics introduced prior to compilation. This artificial intelligence technology appears to be critical for the effective reuse of software across processors.

A major issue is the availability of validated compilers for avionics target processors. Under the auspices of United States Armed Forces programs (i.e., the Ada Integrated Environment and the Ada Language System), compilers have been developed for the following processors:

- 1750A (Air Force)
- MC68000, NS32000, 80X86 family
- AN/UYK-43, AN/UYK-44, AN/AYK-14 (Navy)

While compilers are continually being developed, greater support for typical embedded avionic processors is required. Integration of environments for object-oriented development, code generation, target-specific tailoring, and compilation will be difficult in the near-term. However, as the environments themselves become products of object-oriented Ada development, ease of integration will be a natural consequence.

One final support issue is the target processor execution environment. There are concerns about the real-time capability of the executive programs and run-time services. There are also concerns about the non-homogeneous interfaces provided by these components. While there is only limited real-time Ada execution experience at this time, there is reason to believe that several factors support its practicality: increasing speed of hardware; improving efficiency of compiled code; and application of proven design approaches. In particular, there is nothing inherent in Ada to preclude the use of established deterministic, synchronous design techniques that are typically employed in real-time software. The inconsistent execution environment services and interfaces pose a more difficult problem. However, the essence of the solution lies in the target processor tailoring approaches discussed above. In addition, the various Ada environment communities are aware of this issue, and are working toward standards that will eliminate the problem in the future.

3.4 A Near-Term Prototype

While work remains to be done, the preceding sections show that substantial progress has been made. By the end of 1988, CTA expects to have an operational version of the object-oriented development system, including the following capabilities.

- object-oriented analysis and synthesis
- structured analysis requirements interface
- design elaboration by prespecifications
- design elaboration by functional programming
- automated Ada code generation
- phylum-based reuse environment

The work currently under way concerns integration of the Ada code generation capabilities developed by CTA under separate research projects.

To complete the prototype environment, CTA is in the process of selecting a suitable avionic software development project, for which commercial target processor support is available. In this way, an object-oriented Ada development effort could be pursued during 1989 with no additional environment enhancements required. CTA is also involved in the evaluation of a number of vendor products that could be integrated into the environment as part of the prototype effort.

4. MILITARY AVIONICS APPLICATIONS

This section discusses benefits that would result from the application of an automated code generation environment, such as described in section 3, to military avionics programs. Avionic software, while conceptually similar to other types of software, has typically suffered from the following shortcomings:

- ineffective use of related heritage
- labor-intensive code development
- long concept to validation interval
- high maintenance activity costs

These topics are addressed below, by first describing the nature of the avionics software problem, followed by examining benefits of the automated environment.

4.1 Effective Heritage Reuse

There are certainly differences among airframes, missions, and armaments. The focus seems to be on these differences, rather than on the many similarities. One possible explanation is the acquisition process itself, whereby the platforms, and even models within a given platform series are manufactured by different prime contractors and subcontractors. This leads to a highly competitive arena, with diverse developers, in which there is a lack of developmental history and sharing.

A less obvious explanation, perhaps, is the consequence of functional versus object-oriented development. The functional approach typically evolves along the lines of related software process aggregation; this tends to involve a large number of interfaces between software functions and hardware devices. Changes to the avionic hardware suite, therefore, effect much of the software design.

The proposed development environment would provide benefits with respect to both of the above issues. Object reuse and the phylum object hierarchy provide a mechanism for starting with the general and tailoring to the specific. Coupled with the inheritance concept, this would allow reuse of any existing relevant objects, with addition of new and specific capabilities. Ada language features, with deferred target-processor tailoring, provide an opportunity for increased code reuse. And the object-oriented approach would reduce the impact of subsystem changes, by localizing their effects within the related objects. A final improvement would be realized if the Department of Defense were to incorporate all avionic program developments into a master phylum object data base, and provide it to every contractor. In this way, heritage reuse across all avionic development efforts could be achieved.

4.2 Increased Productivity

Code production is a labor-intensive process. In the case of avionics software, this is often exacerbated by the use of low-level languages. Although the use of high-level languages does improve this somewhat, the process itself is still time consuming. In most projects code production will account for about ten percent of the total labor.

The immediate benefit of the proposed environment is the automated production of code. While this is especially true of generative techniques, late target processor binding allows for increased use of existing code by constructive techniques as well. Perhaps the single most common objection to automated code production is efficiency; of course, this is also the most common objection to the use of high level languages. The response in the same to both objections: first, hardware capacity increases are making this a moot point; and second, it has been shown that selected optimization after implementation is the most effective and cost-effective way to achieve the required performance.

4.3 Rapid Concept Validation

Avionics software development typically exhibits long time intervals between concept formulation and validation. In part, this is a consequence of the lack of heritage and manual code production issues discussed above. In addition, budget and schedule rarely allow for extensive proof-of-concept phases that employ language and execution environments different from the target processor.

Inherent in the proposed environment is the capability to rapidly develop prototype systems. Genericity, abstraction, and reuse facilitate the construction of operational software from the data base of previously defined objects. The Ada code would be executed initially on the host, with target processor binding at a later time. In fact, automated target-specific tailoring would allow testing on similar hardware prior to availability of the actual avionic equipment. Early focus on similar features, tailoring of unique aspects, and late target processor binding support rapid concept implementation and validation.

4.4 Enhanced Maintainability

Entropy has a tendency to degrade the maintainability of all software. Military software also suffers from being removed in distance from the maintenance organization. Unfortunately, this seems to accelerate the deterioration of both the software and the associated development documentation.

The proposed environment does not offer any maintenance features that are unique to avionics software. However, any improvement in the maintenance process will result in significant cost savings. The object-oriented development process does offer several benefits. One of these is the ability to work at higher levels of abstractions. This reduces the overall scope of the modification process. Since code is an end-product derived from the overall process, not an independent development step, all changes start with the specification, and propagate through to the code. The result is that the software maintenance process also maintains the integrity of the object hierarchy. The ultimate benefit is consistent and complete specifications, with highly leveraged labor, in contrast with maintenance efforts that focus on code level modifications.

5. SUMMARY AND CONCLUSIONS

This paper suggests that, as a consequence of the coming Ada imperative and the increasing quantity and complexity of avionic software, improved development methodologies are needed. CTA has addressed such innovations, and has experience that demonstrates their effectiveness. An approach to an automated Ada development environment has been proposed, of which many elements exist today and others are being added.

A pilot avionic software development project is feasible within the next year, using the proposed environment. The paper has discussed application of the methodological principles to Ada code generation. While Ada supports these object-oriented ideas, and is intended for widespread use within the defense industry, it should be noted that the automatic programming mechanisms are not unique to Ada, and could be used to target other languages.

REFERENCES

[1] J. Tannenbaum, Salomon Brothers Inc., "The Military's Computing Crisis: The Search For a Solution", September 22, 1987

[2] Institute of Electrical and Electronics Engineers Inc., "IEEE Recommended Practice or Ada As a Program Design Language", 1987, Standard 990-1987

[3] G. Booch, "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, Pp 211-221

[4] M. Jackson, "System Development", Englewood Cliffs, New Jersey, Prentice-Hall, 1983

[5] J. Martin, "Application Development Without Programmers". Englewood Cliffs, New Jersey, Prentice-Hall, 1982

[6] P. Baker, "Partial Automation of Code Generation with Ada Prespecifications". Proceedings of the Joint Ada Conference, Washington, District of Columbia, March 16-17, 1987, p. 397

[7] T. DeMarco, "Structured Analysis and System Specification", Englewood Cliffs, New Jersey, Prentice-Hall, 1979

[8] A. Goldberg and D. Roboson. "Smalltalk-80: The Language and Its Implementation", New York, New York, Addison Wesley, 1983

[9] E. Seidewitz, "Object-Oriented Programming in Smalltalk and Ada", ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, Pp. 202-213

[10] G. Booch, "Software Components With Ada - Structures, Tools, and Sub-systems", Menlo Park, California, Benjamin/Cummings, 1987

[11] S. Bailin and J. Moore, "A Software reuse Environment", Proceedings of the 12th Annual Software Engineering Workshop, NASA Goddard, Maryland, December 2, 1987

[12] Department of Defense, "Military Standard - Defense System Software Development", February 29, 1988, Standard 2167A

DISCUSSION

M.Muenier, Fr

According to your presentation, it seems that a whole environment covering requirements and design specification and coding is not yet available. Would you comment on the status of your project and the planned development?

Author's Reply

For both automatic programming techniques, the "tool" is essentially an environment for employing the software components developed by a domain expert.

In the case of Ada prespecifications, the domain expert specifies a domain-specific metalanguage (i.e., for graphics, data management, device class control, etc.). For *each* metalanguage construct he then develops an Ada body generator procedure (in Ada). The "tool" provides the mechanisms for users to develop their metalanguage specifications and to invoke the appropriate code generation procedures.

In the case of functional programming, the domain expert would develop a set of Ada code body functional primatives. The "tool" provides a graphic interface to construct data flow diagrams from the functional primatives, enforces certain conventions to eliminate possible ambiguities and constructs the code body procedure invocations necessary to "implement" the specified diagram.

W.Mansel, Ge

Can you add some comments about the tools developed and used for the design specifications and the code generation from a library of Ada code?

Author's Reply

At the present time, we have a core of Object-Oriented Development (OOD) tools. Ada prespecifications (including the Phylum Mechanism) have been demonstrated and are being incorporated into the OOD toolset this year. A rudimentary demonstration of the functional programming concept has been accomplished; this year we are extending (generalizing) the concept and incorporating it into the OOD toolset as well. When complete, this will provide an integrated toolset capable of supporting all activities up through execution of general (host) Ada programs.

The target processors specific tailoring, cross compilation and run time binding will be provided by a commercial product for a selected application. We have no plans to attack the expert system approach to target processor tailoring at this time, although we are following some on-going work by other groups.

The reuse environment is under development as part of a NASA Goddard Contract. When complete, it will also be incorporated into our OOD toolset. The toolset currently runs on a Sun workstation. We are in the process of transporting it to a DEC Vax station.

VERIFICATION AND VALIDATION OF FLIGHT CRITICAL SOFTWARE

by

Pio de Feo
Director, Software and Systems Technology
Sparta, Inc.
Laguna Hills, CA 92677
USA

Anthony DeThomas
Senior Engineer
AF Wright Aeronautical Laboratories
Flight Dynamics Laboratory
WPAFB, OH 45433
USA

SUMMARY

The complexity and the fault tolerance requirements of flight critical systems are rapidly increasing. New techniques and methods must be developed and must be integrated with existing methods so that the rigorous technical objectives of the verification and validation process of such systems can be met at reasonable cost. This paper examines some of the advanced requirements of these systems in the critical areas for system architecture and software design. Some promising techniques are described which can effectively support architectural design and analysis and software development and verification.

1.0 INTRODUCTION

The use of digital technology in flight critical applications has permitted substantial improvements in vehicle and mission performance because of the flexibility of design and implementation it provides. The requirements for ever increasing functional performance and reduced pilot workload have resulted in complex flight control system architectures and communications schemes and correspondingly complex software. Such systems often are safety critical during some flight regimes and, in some cases, during the entire flight. Redundancy and advanced fault tolerance techniques are needed to meet the safety requirements of those systems. The resultant complexity of system architecture and embedded software is such that the current verification and validation techniques and methods may not be technically adequate and cost effective. In this paper some critical technology needs are described for supporting the development and verification process of the highly integrated flight critical systems of the next generation of aircraft. Two methods are also described, which if appropriately integrated with existing technology, can satisfy, at least in part, those needs.

2.0 SYSTEM REQUIREMENTS

The functional requirements and the architectures of automatic flight systems are continuously evolving (1), see Fig. 1. Limited authority, analog, stability augmentation systems (SAS) were developed during the 1950's and were followed by the flight critical analog flight-by-wire (FBW) systems during the 1970's. The development of flight critical digital FBW (FCD-FBW) systems started in the early 1980's, and it is still evolving. Flight critical FBW, in this context, means that loss of that system results in catastrophic consequences for the aircraft and possibly the pilot. The trend is clearly established towards systems which are increasingly complex and critical. The evolution and ensuing complexity of flight control systems have been accelerated by the need to increase aircraft dynamic and operational performance. Specific contributors to this trend are:

a) The development of multimode control laws which permit the tailoring of the vehicle dynamics and handling qualities to specific mission phases, such as, takeoff and landing, cruise, air-to-air combat, and air-to-surface attack. Utilization of task tailored control laws allows the enhancement of flying qualities in particular mission segments without compromise to other mission segments.

b) The integration of flight control with other aircraft and mission functions which improve vehicle performance and total aircraft system efficiency and decreases pilot workload. Examples of such systems are the integration of flight and propulsion control, fire control, and inertial navigation systems. Control and avionics functions are being coupled to satisfy advanced mission requirements, such as terrain following/terrain avoidance (TF/TA) missions. These applications require sensory functions such as terrain data, radar altitude, target acquisition and tracking, and inertial reference system data, in addition to flight control functions. The integration effort may be further expanded to include control of utility functions such as electrical and hydraulic power, environmental control and fuel management. Such systems are referred to as vehicle management systems (VMS).

c) The need to meet stringent safety-of-flight requirements and the corresponding reliability, availability and fault tolerance requirements. In some cases, the boundaries of flight critical control functions are growing beyond the classical control systems applications discussed above, as in the case of highly integrated military applications. As an example, missions such as TF/TA and nap-of-the-earth (NOE) must be considered not only mission critical but flight critical since loss or malfunction of these systems may result in catastrophic consequences.

The flight safety, reliability requirements of critical flight control systems is not to exceed 10^{-7} catastrophic failures/hour (F/H) in the case of military aircraft, and 10^{-9} catastrophic F/H, in the case of commerical aircraft. The failure rates of typical flight control components exceed those requirements by several order of magnitude, Fig. 2. It is therefore, necessary to design redundant configurations so that flight safety failures can only occur as a result of multiple failures.

The quantitative requirements for flight control system flight safety are given in MIL-F-87242 (2) for US military systems. The guidelines for the certification of commercial aircraft are described in the Code of Federal Regulation (CFR)(3). The CFR states that system failures which prevent continued safe flight and landing of the aircraft must be extremely improbable. The CFRs do not define a maximum acceptable frequency of such failures. The normally accepted interpretation of extremely improbable is, however, a frequency of occurrence not higher than 10^{-9} F/H, as previously stated. Similar guidelines also exist for commercial aircraft certification in all NATO countries.

It must be noted that the failure rate requirements refer to the hardware/software combination. During the system development process, however, the assumption is often made that the software does not contribute to the system failure rate. The software is assumed to be error free through a combination of stringent development and verification techniques and other design considerations as described later in this paper. This approach is at least in part justified by the fact that the current software technology does not support a-priori quantitative assessment of the probability of residual errors. The final outcome is that, in most cases, the system level reliability is met by providing the appropriate level of hardware redundancy only. In some cases, as further discussed later in this paper, dissimilar software is also used to provide some level of software redundancy. The purpose is, however, to achieve software fault tolerance rather than a quantifiable and demonstrable improvement of software reliability.

In the following sections of the paper, the characteristics of the software required for such systems and the software development and verification process will be described.

3.0 SOFTWARE CHARACTERISTICS

To determine the required characteristics of flight software, it is necessary to consider the control system characteristics and the implications on mechanization of the software (4). Table 1 summarizes these characteristics and resultant software implications.

The organization of the embedded flight software is by design very simple given the critical and complex nature of the application. It implies the repetitive execution of application tasks, which once initiated, are non-interruptible and must execute to completion within a predefined period of time. To gain a better understanding of the computational requirements of such systems, the software is partitioned according to the major functions to be performed. They are:

a) Application. The algorithm included in this function are those required for sensor processing and filtering, control and navigation algorithms, and computation of control commands.

b) Logic. Modules in this category perform the computation required for switching control and flight mode, and engaging/disengaging logic. They use almost exclusively boolean statements.

c) Testing and Voting. These modules perform real time test on processors, memory, sensors and actuators. They manage and control the overall system configuration as a function of detected failures. Built-in-test (BIT) is included in this category.

d) Input/Output. These modules perform data handling and formatting, data transmission and display. Peripherals drivers are included in this category.

e) Executives. These modules perform the task of initialization, power-up procedures, synchronization, scheduling and timing.

The typical, memory requirements of each software function, as a percentage of the total, are shown in Table 2. The data shown in the table represent an average of several systems for military and commercial aircraft. Clearly, differences do exist from case to case. As an example, in the case of a system which relies significantly on self test for the purpose of failure detection and isolation, testing does take a larger percentage of the total memory than that shown in the table. Complex, highly redundant, distributed configurations, have relatively high logic, testing and executives content. Simplex configurations, with limited integration of functions, tend to have a relatively high application content.

Equally important for understanding the software of critical systems is to analyze the nature of the software errors which are detected during software and system development. A common characteristic of all programs which were analyzed is that the majority of software errors are generated during the early phases of the development process including definition of requirements, and the development of specifications and design. Five general error categories have been identified with the frequency of occurrence of each category in Table 3.

It is important to notice that the table does not include the occurrence of trivial coding errors, which are easily detected during software coding and debugging and even during module testing. The table includes only those errors which were detected while the software was already in configuration control, and therefore had already been extensively tested at the module level and at the level of software integration. Those errors were detected during the software/hardware integration test, iron bird test, and flight test. Most errors, as previously stated, are a result of incomplete or inconsistent requirements, incorrect specifications and/or poor or wrong design decision. The table shows that errors associated with logic, interfaces and data handling, including testing and voting are the biggest contributors. There are some reasons for this to occur. Those areas are in fact rather new, relative to the application functions. The necessary tools are not well developed and the understanding of complex architectures (and therefore complex interfaces), logic and failure detection and reconfiguration algorithms is not as mature and established as that of the application algorithms.

4.0 DEVELOPMENT AND VERIFICATION GUIDELINES AND SUPPORTING ENVIRONMENT

In recent years, significant progress has been to improve our understanding of the software development and verification process. The major result has been to move the emphasis from coding to design, and to enforce methods which produce simple, easy to understand software. Equally important has been the determination that the verification process does not start after code has been developed, but it must parallel the entire development effort from the beginning. In fact, the errors most difficult to detect and costly to remove are generated during the early phases of the development process, as previously discussed. The importance of a structured development and verification approach is particularly felt in the case of safety critical applications like flight control systems. Much documentation exists relative to methods, tools, techniques, and environment which support such processes. Guidelines for military systems are described in the DoD-STD-2167A (5). Corresponding guidelines for commercial systems are described in the RTCA document RTCA/DO-178A (6). Both documents, define an amount of development, verification and documentation efforts which increases with the criticality of the application, and both stress the importance of initiating the verification process at the earliest phases of the development process. Figure 3 shows the progression of software development activities and the corresponding verification activities. The development process includes requirements development, design, coding, integration and system test. The primary supporting documentation to be produced in each phase is also shown.

The software verification assurance metrics for each level of software is shown in Table 4. It is observed that the rigor and formality of the assurance process is the most demanding for critical functions (LEVEL I) which effect flight safety.

The existing V&V technology for advanced FCD-FBW systems, must be continuously enhanced by new tools, techniques and methods in order to maintain the V&V cost within reasonable bounds. Some systems require the development of V&V capabilities not currently available. As an example, the verification of systems capable of performing NOE and TF/TA missions, requires the availability of data from several sensors which cannot be realistically produced in current simulation environments. Appropriate techniques must then be developed so that the cost effective real time, pilot-in the-loop simulated environments can still be an essential part of the development and validation process of those systems. The only available alternate approach is to validate those functions entirely by flight testing, which is a prohibitively expensive proposition.

A typical current software development and verification process is supported by two distinct environments. The first environment, usually hosted in a main frame or mini computer operating in an interactive mode, is applicable to a broad category of digital flight system (DFS) programs and provides general purpose capabilities in the following areas:

1) Control Laws Analysis. Tools are provided which support design and analysis of continuous and discrete control systems using classical and modern techniques. An example is frequency response and root locus analysis to assess the performance and stability of classical control systems. In the case of optimal control systems, eigenvalues/eigenvector analysis is used for the same purpose.

2) Reliability and Failure Mode/Effects Analysis. Reliability analysis is used to evaluate the probability of a system to successfully perform specific tasks. Failure mode and effects analysis (FMEA) establishes the effects of system components failures on overall system performance. The combined results of the two analyses is to verify that any failure, or combination of failures, which could have catastrophic consequences, has only a very low and acceptable probability of occurrence.

3) Software Development. Software tools support several software development tasks, such as software design and documentation, code generation, and configuration management. The tools which support the critical task of code generation are required to be very mature; compilers, assemblers and linkers fall in this category.

The second environment has the capability of exercising the flight software in real-time. This environment is typically developed for and dedicated to a single program. It includes essential elements of the flight control system (FCS), and the airplane dynamics, so that closed loop analysis can be made. Some FCS components are simulated, some are emulated and others are directly included in the environment. Typically, the flight computer hardware is included so that flight software can be exercised in a representative environment. The simulation of the aircraft dynamics and of those FCS hardware components that would be impractical to include in the environment are implemented in a dedicated host processor linked to the flight computers. These real-time environments play an extremely important role in FCS development.

The entire software V&V process is affected by: a) the early selection of the design and development approach, and b) the availability of advanced tools and methods. Both topics are further expanded in the following sections of this paper.

5.0 DESIGN CONSIDERATIONS

The software issues which mostly affect the V&V process are: a) the use of high order languages (HOL) vs. assembly language; b) the use of dissimilar software; and, c) the methods for supporting the early phases of the design process. They are discussed further in the following paragraphs.

5.1 HOL vs. ASSEMBLY

The use of HOL is increasingly accepted in flight applications. There are many reasons for increasing use of HOL:

a) Compilers are getting more efficient. The memory and time penalties associated with the use of HOL instead of assembly are becoming smaller (50% and 20% respectively), and less significant due to increasing computational speeds and decreasing cost of memory.

b) There is a strong indication that HOL programs are more reliable than assembly programs. In a study of software reliability of FCS it was found that the fault density of FCS software written in HOL is approximately 30% of that of assembly program (7).

c) HOL programs are easier to develop, test, understand, modify and maintain than assembly programs.

d) Significant experience is now available relative to the performance and the code generated by mature software tools, like compliers, assemblers and linkers.

For all the above reasons the V&V process of flight software is simplified a great deal if the programs are written in HOL rather than assembly.

5.2 DISSIMILAR SOFTWARE

The purpose of using dissimilar software, in redundant configurations, is to prevent catastrophic consequences from a common, undetected error, in all channels. Dissimilar software can be used in two ways:

a) As a failure detection mechanism in the operational system. In this case, both software versions execute at the same time. The results are compared periodically, and in the case that the results differ by more than certain values, a fault condition is detected. In that case, reversion to a back-up control mode is made, which does not utilize either of the two software versions.

b) As a back-up control mode. In this case, if a common failure is detected in the primary software, reversion is made to the alternate version.

5.3 THE EARLY PHASES OF THE DEVELOPMENT PROCESS

A clear evidence exists that the software errors which are most difficult and costly to detect are often introduced early in the development process. This points to the need for tools, techniques and methodologies capable of supporting effectively the specification, and design phases.

It is extremely important to define, as early in the development cycle as possible, design disciplines which make the software traceable, testable, maintainable and easy to understand. Design and coding standards must also be established. Examples of standards commonly enforced are: a) limiting the maximum size of the smallest software component, the module, b) avoiding constructs which make it difficult to understand and verify the code, like unstructured GO-TOs, and c) providing clear and comprehensive documentation embedded in the source code listings.

6.0 ADVANCED TOOLS AND METHODS

The techniques described in this section address some of the most critical technology needs in the areas of system architecture design and analysis, and software development and verification. They are intended to be integrated with existing and new technologies in cost effective and easy to use environments.

6.1 SYSTEM ARCHITECTURE ANALYSIS

The selection of a DFS configuration is the result of difficult compromises among several, and often contrasting requirements and constraints. The designer needs tools for simulating and analyzing the relative merits of competing configurations, very early in the development cycle, prior to committing to any specific configuration. Two different methods of analysis are proposed. The first method, is based on the complexity analysis of the topography of the architecture, and of the corresponding structure of the software. The second method, is based on a non real-time simulation of the operational performance of the system under a variety of environmental conditions.

6.1.1 COMPLEXITY METRICS

Complexity metrics can support software code, software design and system architecture analysis. An approach is the graph theory based McCabe's metrics, which calculates a Cyclomatic Number (CN) from the difference between the number of nodes and the number of edges (8). A node is a straight line segment of code, a functionally cohesive software module or an hardware embedded function, in case of software code, software design, and system architecture analysis, respectively.

When the McCabe's model is applied to DFS architectures, the real-time execution of the software embedded in each processor must be considered. Factors which must be considered include:

a) Flight software executes at different rates, depending on the dynamics of the control algorithms.

b) Advanced, multimode, flight control systems perform different tasks depending on the aircraft flight and mission mode.

c) Some low priority application modules can be interrupted, which increases the complexity of the control flow.

d) Distributed architectures which have a degree of complexity higher than a single processor architecture because they require additional communication for exchanging data, time synchronization, failure detection and reconfiguration.

The architecture and software complexity metrics are static tools which can be applied throughout system design phases in a manner that permits the inclusion of more design details as they become available. They can be effectively utilized during the early development phases for comparing competitive architectures and analyzing the effects of design modifications.

6.1.2 SIMULATION TECHNIQUES

Another important measure of the quality of the architecture can be achieved from dynamic parameters such as the distribution of the computation load among processors, the utilization of, and competition for, shared resources such as common busses, and other time sensitive parameters. The parameters can be estimated by simulating, in a non real-time environment, the relevant sequences of tasks to be performed, and by keeping track of the time needed by the allocated resources to perform those tasks. To generate this data, the environment must be able to simulate: procedural processes, such as the highly structured flight control algorithms which execute under rigid, predetermined time sequences; parallel processing, as in the case of redundant, distributed configurations; shared resources, such as common busses; and the executive logic which controls the execution flow within each processor in those cases which require detailed analysis of the time sequences of events . The output of these performance simulation runs are: statistics of resource utilization and detailed time sequence events like the exact timing of data exchange among redundant processors.

Most of the elements included in the performance simulation environment just described have been developed for, and effectively used in, application such as Command, Control and Communication (9). This type of simulation has not been applied to DFS because of the relatively simple configuration structures of current systems. On the other hand, the architectural complexities of the next generation DFS justifies the need for those capabilities.

The simulations and the complexity metrics tools, described in this section of the paper, can be hosted in the same environments which currently support control law development and FEMA analysis, and software development.

6.2 SOFTWARE VERIFICATION

Software is often the most critical element in terms of development, verification and maintenance cost, and software is a major contributor to the overall failure rate of current DFS. The current, state-of-the-art, software test strategies are structured to exercise all independent software logical paths with a minimum set of test data. A removal of all implementation faults, however, cannot be guaranteed even if 100% test coverage is achieved. In fact, failures can still occur as a result of unusual logical or timing sequences of events. Then if one of these combinations should occur in flight, a system failure could result with unpredictable consequences.

To achieve a high level of confidence that all the failure triggering combinations are detected, a stress test technique is described which is a highly automated, very effective way to test code with an extremely large number of test data sets while it is executing in the target processor(s). Several properties of the software can be stress tested, among them: the computational algorithms, the timing requirements and the partitioning existing among modules of different criticality. These topics are discussed in the following paragraphs.

6.2.1 DATA STRESS TEST

This test verifies the correct algorithmic execution of a software application functions (10). Examples are the numerical intensive algorithms like those implementing guidance and controls laws, and logical intensive algorithms, like those found in fault detection/isolation and system reconfiguration.

The essence of this test is to execute critical software functions in the target computer concurrently with the execution of alternate benchmark versions in a host machine. The benchmark is coded directly from the design document. The same exhaustive input sequence of data, within and outside the specification boundaries of each function, are then used to exercise both implementations at the same time. Then errors in either implementation will result in different outputs, under certain test conditions, because it is extremely improbable that the same errors are made in both versions. Because of the extremely large number of test data involved, it is necessary to automate the entire stress testing process.

6.2.2 TIMING STRESS TEST

This test measures the statistical distribution of the computational and transport delays between the time variables are set in one processor and the time they are used in other processors; and it verifies that the delays do not exceed predetermined maximum values.

The two main factors to be considered in the application of this test are:

a) Intraprocessor Delay - The computational delays between the time the updated values of variables are available within a processor, and the time the outputs of the algorithms using those variables, are actually computed. This delay is a function of the computational logic path, within the processor, which can vary from iteration to iteration. Software multirate structures, changes of aircraft flight mode or cockpit display mode and changes of some parameters values can affect these delays.

b) Interprocessor Delay - The transmission delay between two processors including: 1) the computational delay between the time the output of an algorithm is updated within a processor and the execution of the output statements; and 2) the time needed to transfer the actual data from the output buffer of one processor to the input buffer of another processor, inclusive of any buffer access and transmission times. These components vary as a function of the execution paths of the two processors and of the availability of shared resources such as bus interfaces.

The ultimate objective of timing stress test is to determine, for each time critical variable, the distribution of computational and transport delays; and verify that, under no situation, those delays do not exceed predetermined maximum values. The entire process of time stressing can be easily automated using the same environment which supports data stress test (10).

6.2.3 PARTITIONING STRESS TEST

This test verifies the partitioning between two functions. Function "A" is partitioned from "B" if not, action of function "B" can cause a failure of function "A". The need for this test derives from two considerations: 1) DFS configurations include functions and components of different levels of criticality; and 2) the required development effort and development cost vary a great deal as a function of the criticality. If partitioning cannot be demonstrated between functions "A" and "B" with different levels of criticality then both functions must be considered as having the same level of criticality which is the highest between levels "A" and "B". This unduly increases the development and verification cost of the functions which have the lowest intrinsic criticality level.

The objective of partitioning stress test is to demonstrate partitioning between two functions of different levels of criticality. To achieve this objective it must be demonstrated that no output data from the least critical function, and no time skewness between the execution times of the two functions, can cause a failure of the most critical function.

The currently used real-time, dedicated environment which includes the target computers and embedded software is an ideal host for the stress techniques previously described. Requirements are that the software under test execute under conditions which are representative of the flight conditions.

7.0 CONCLUSIONS

The software characteristics and the existing guidelines for the development and verification of flight critical systems have been described. The technology requirements for flight critical systems for the next generation of aircraft have also been analyzed. Major technology needs have been identified in the areas of system architecture design and software development. Two methods to satisfy these requirements are described which can be highly automated and can be integrated with an existing development environment. The first method leads to a quantitative assessment of the quality of competing system architectures. The second methods can be utilized for testing some critical software requirements with extremely large sets of test data. Both methods have the promise of significantly enhancing the technical capability of the existing development and verification procedures and of reducing the life-cycle costs of the next generation FCD-FBW systems.

REFERENCES:

1. R.A. Rediess and E.C. Buckley, HR Textron Inc., Technology Review of Flight Crucial Flight Control Systems, Apr 84, NASA Contractor Report 172332

2. MIL-F-87242, "Military Specification Flight Control Systems, General Specification For", 31 Mar 86

3. Code of Federal Regulations, Part No. 25

4. D.B. Mulcare, et al. INDUSTRY PERSPECTIVE ON SIMULATION METHODS AND RESEARCH FOR VALIDATION AND FAILURE EFFECTS ANALYSIS OF ADVANCED DIGITAL FLIGHT CONTROL AVIONICS, Feb 79, NASA Contractor Report CR-152254

5. DoD-STD-2167A, "Military Standard, Defense System Software Development", 29 Feb 88

6. RTCA/DO-178A - "Software Considerations in Airborne Systems and Equipment Certification", Mar 85

7. Hecht, H., Hecht, M., Trends of Software Reliability for Digital Flight Control, Apr 83, NASA Contract Report P.O. A93024B

8. McCabe T.J. "A Complexity Measure" IEEE Transactions on Software Engineering, Vol. SE-2, NO. 4, Dec 76

9. Kneeburg S., Automated Interactive Simulation Model (AISIM) Contract F33615-81-C-5098; Electronic Systems Division, Hanscom Air Force Base, MA

10. Rajan N., De Feo P. et al., "Stress Testing of Flight Control Systems Software IEEE/AIAA 5TH Digital Avionics Systems Conference, Nov 83

11. De Feo, P., DeThomas, A., "Technology Requirements of Integrated, Critical Digital Flight Systems" AIAA Guidance, Navigation and Control Conference, Aug 87

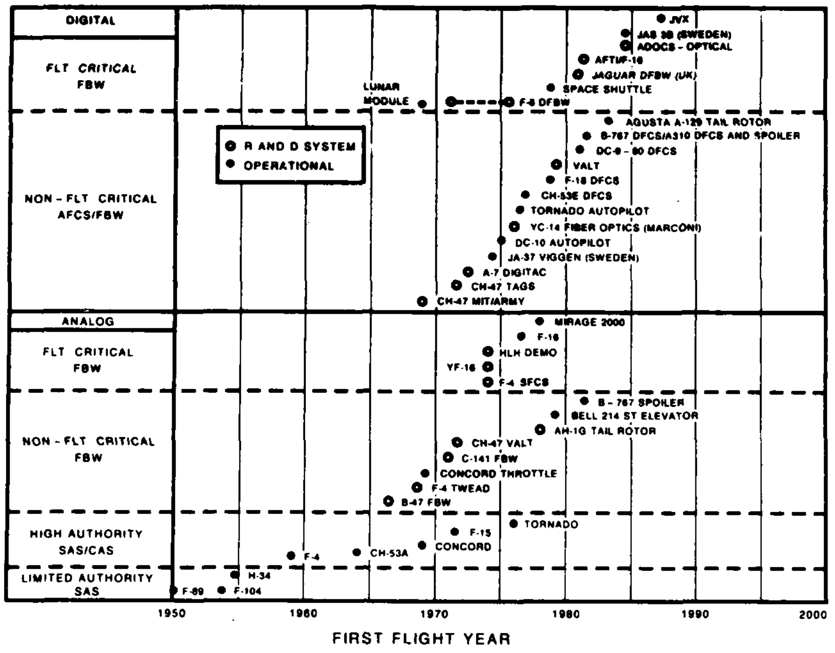


Figure 1. FLIGHT CONTROL SYSTEM EVOLUTION

CHANNEL ELEMENT	MTBF
RATE GYROS	20,000 EA
ACCELEROMETERS	30,000 EA
ATTITUDE HEADING REFERENCE SYSTEM	1,300
DIGITAL AIR DATA SYSTEM	4,000
FLIGHT CONTROL DIGITAL COMPUTER	4,000
INPUT/OUTPUT	
MUX AND A/D	70,000
DEMUX AND D/A	70,000
SERVOS	8,000

Figure 2. TYPICAL FLIGHT CONTROL SYSTEM COMPONENT FAILURE RATES

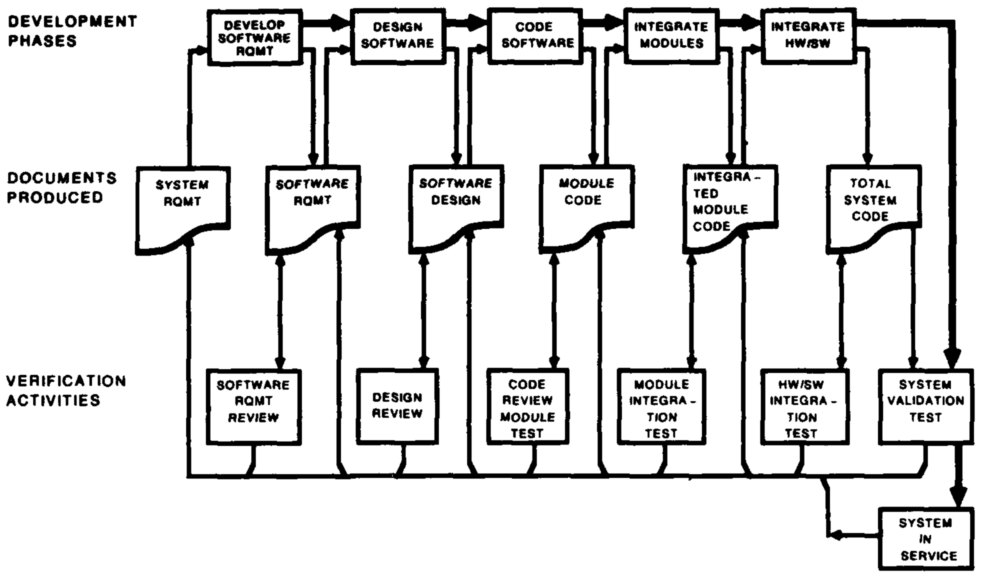


Figure 3. SOFTWARE DEVELOPMENT AND VERIFICATION ACTIVITIES

Table 1. DIGITAL FLIGHT CONTROL SYSTEM CHARACTERISTICS

APPLICATION CHARACTERISTIC	MAJOR CONSIDERATION	RESULTANT SOFTWARE IMPLICATIONS
EMBEDDED COMPUTER(S)	SYSTEM FUNCTIONS & INTERFACES	MULTIPLE TASKS, I/O HANDLING, SYSTEM MONITORING
REAL-TIME OPERATION	CAPACITY & TIMING	THROUGHPUT, START-UP, SYNCHRONIZATION
CLOSED-LOOP OPERATION	MINIMAL DELAY	INLINE PRIORITY
WIDE BANDWIDTH	ITERATION RATE	THROUGHPUT, MULTIRATE SAMPLING PARALLEL PROCESSING
HIGH FUNCTIONAL RELIABILITY	REDUNDANCY	INTERCHANNEL COMMUNICATION, EQUALIZATION
SAFETY CRITICALITY	INTEGRITY	SELF-TEST, INLINE MONITORING, DISTINCT SOFTWARE
CREW INTERACTION	HUMAN INTERFACE	CONTROL/DISPLAYS, PILOT COMMANDS, STATUS ANNUNCIATION
OPERATIONAL ENVIRONMENT	ADAPTABILITY	VARIABLE FLIGHT DYNAMICS, ATMOSPHERIC CONDITIONS, PERFORMANCE MONITORING, MODE CHANGES
PHYSICAL ENVIRONMENT	NON-SUSCEPTIBILITY	SENSOR NOISE, EMI, POWER VARIATIONS

Table 2.
MEMORY REQUIREMENTS FOR FCS SOFTWARE

SOFTWARE FUNCTIONS	MEMORY REQUIREMENTS
APPLICATION	25%
LOGIC	20%
TESTING	20%
I/O & EXECUTIVE	20%
MISCELLANEA	15%
TOTAL	100%

Table 3.
FCS SOFTWARE ERROR DISTRIBUTION

ERROR CATEGORY	FREQUENCY
COMPUTATIONAL	10%
LOGIC	25%
DATA HANDLING	35%
INTERFACES	10%
MISCELLANEA	20%
TOTAL	100%

Table 4. SOFTWARE VERIFICATION ASSURANCE MATRIX

	CRITICAL			ESSENTIAL			NON-ESSENTIAL		
	LEVEL 1			LEVEL 2			LEVEL 3		
VERIFICATION/ TEST ACTIVITY	A	I	S	A	I	S	A	I	S
VERIFY SOFTWARE REQ. vs. SYSTEM REQ.	X	X				X			
VERIFY DESIGN vs. SOFTWARE REQ.	X	X				X			
VERIFY CODE vs. DESIGN		X				X			
TEST SOFTWARE MODULES									
TEST MODULE INTEGRATION	X	X		X	X				
TEST HARDWARE/SOFTWARE INTEGRATION									

DISCUSSION

A.O.Ward, UK

Do you know of any projects that have adopted Ada for safety critical systems?

Author's Reply

I am not aware of any current production flight critical systems which are utilizing Ada.

J.P.Lacroix, Fr

Do you agree that software dissymetry should be extended to development and include two different teams?

Author's Reply

The utilization of dissimilar software is a means for improving the fault tolerance of the software within the system. The use of two independent development teams would greatly improve the reliability of the fault critical system if the process was in fact independent. This implies that independent specifications are developed since independent development using a common requirement could carry the same design errors through both systems. Although the two independent development teams approach was an advantage, however, the cost would probably prohibit its use.

W.Mansel, Ge

When using HOL for flight critical software, a verification of the compiler has to be considered. Do you assume that this is covered in your verification by the code verification process which also includes the verification of the run time system?

Authror's Reply

Errors introduced by compilers have always been a concern. However, in our experience, although limited at the present time, we have not found this to be a problem. The rigorous V&V process used for these systems has detected compiler errors and permitted correction.

Guidelines for Evaluation of Software Engineering Tools

Robert E. Marmelstein, 1Lt, USAF
Computer Research Scientist
AFWAL/AAAF-2
WPAFB, OH 45433, USA

I. Introduction

For better or worse, the way military software is developed and maintained will undergo drastic changes in the next decade. This is a foregone conclusion since most nations cannot afford the status quo. "Annual software costs for major defense systems, which were at about $4 billion in 1980, are now at approximately $10 billion, and are expected to reach $30 billion by the early 1990s." [1] This projection is driven by the increasing need for software in weapons systems, as well as, the increasing cost of developing and maintaining this code.

The cost of software has forced the USAF to explore new types of programming environments to develop and maintain software. "A programming environment is the hardware and software required to support software life cycle activities. These life cycle activities include concept definition, requirements analysis, preliminary design, detailed design, program development, software integration, system integration and testing, and maintenance." [2] A typical programming environment consists of tools which are used in combination to develop software; a barebones environment consists of a text editor, compiler, linker, and a debugger. In the last few years, USAF interest has spawned the development of software engineering tools to address every phase of the programming lifecycle (from requirements analysis to maintenance). Although these tools represent an important step in automating the software lifecycle, there is often confusion surrounding the selection of the proper tool for a user's application. This often results in tools being selected which does not properly meet the user's needs. The principal causes of this situation are:

-- *The capabilities of the tool are incomplete or inadequate.*
-- *The choice of a tool is premature.*
-- *Problems arise with tool portability.*
-- *The tool is overly complex.*
-- *The tool is not robust.*

Because the mandated use of an inadequate software engineering tools in the software lifecycle can have devastating effects on program costs, the need for a taxonomy to evaluate these tools is readily apparent. This taxonomy should contain guidelines which are applicable to the design of a new tool or the selection of an existing one. This paper will define a basic taxonomy for evaluating software engineering tools.

II. Description of the Interactive Ada Workstation

The Interactive Ada Workstation (IAW) is a prototype programming environment for the Ada language. It combines a number of state-of-the-art capabilities such as graphical design techniques, automatic code generation, incremental code analysis, incremental debugging, and reusable software aids in order to significantly increase the productivity of the average programmer. The IAW is hosted on a Symbolics 3670 Lisp Machine; this host was chosen for its to specify the top level, data flow design of his program. The design is expressed in terms of icons which represent packages, subprogram units, data types and their interfaces. From this description, the corresponding Ada code will be generated.

b. **State Machine Editor:** The purpose of this editor is to provide the programmer with the ability to specify the top level control flow for a particular subprogram unit (task, procedure or function). The design is expressed in terms of states (representing specific actions to be taken) and transitions (representing control flow decisions) between states. Once a design has been generated, the programmer can use a simulator to test his model. This description can be used to generate the Ada code template for the desired subprogram.

c. **Decision Table Editor:** This editor has the same general purpose as the State Machine editor, however, it makes use of a different design methodology to accomplish its purpose. The Decision Table editor (DTE) allows the programmer to define rules and attach specific actions to them. The Decision Table editor has a tabular spreadsheet format, rather than the icon based format of the BRAT and State Machine editors. Like the previous editors, the DTE can generate the Ada code corresponding to its description.

d. **IAda Editor:** The IAda editor is a syntax directed editor for the Ada programming language. Its purpose is to give the programmer the capability to fill in the code templates generated by the graphical editors. The IAda editor will also perform incremental syntax and semantic analysis on the Ada code within its text buffer (syntax errors are immediately flagged with inverse video in the text buffer and semantic errors are logged in a separate window). The

IAda editor will build an internal representation of the program (called IForm) as it is being incrementally analyzed. Once the Ada program has been written using this editor, the programmer can then regenerate the updated BRAT diagram from the IForm. In future versions of the IAda editor an interpreter will be provided to allow the programmer to test his code.

The purpose of the IAW program was to provide a proof-of-concept that these state-of-the-art programming techniques could be implemented together on a high performance engineering workstation. As a result of this program, General Electric (developers of the IAW) have teamed with Cadre Technologies to market the IAW as a product. The IAW project also gave AFWAL a considerable amount of insight into the attributes an advanced programming environment should have.

III. Evaluation Methodology

a. Host Computer Considerations

The selection of a host computer is one of the most important decisions to make concerning the development of software. This section will explore some of the ramifications of host computer selection upon programmer productivity.

(1) Cost

The cost of the host system should not be prohibitively high in terms of a cost/user ratio. The advent of powerful personal computers and engineering workstations gives the individual increased computing power at considerably less expense. A Sun 3/50 Workstation (which is based on a Motorola 68020 processor running at 16MHz) can be purchased for under $8K. As a result, these systems are becoming increasingly popular. A figure of $7,500/programmer is a reasonable cost/user ratio.

(2) Vendor Support

The host computer vendor must be available to give adequate support. The software developer should be concerned about support in the areas of user support (documentation, hot-lines, etc), software support (availability of quality software and documentation), and hardware maintenance (quality and timeliness of system maintenance). Without proper vendor support even a state-of-the-art computer can quickly become obsolete.

(3) Availability

Availability can be viewed in two ways. The first is availability within an organization. A programming tool will not improve productivity if personnel have to wait in line to use it. Availability is one of the greatest benefits of timesharing computer systems. Individual workstations (such as the Sun) lose their appeal if there is a high user/machine ratio. The second way of looking at availability is how widespread is the use of a given system. A host system which is powerful, but quite expensive (a Lisp Machine, for example) will not be available in volume to other organizations. This limits the ability of the software developers to transition a program once it has been developed on a specific environment.

(4) Host/Target Computer Relationship

The number of computers needed for software development should be limited as much as possible. Ideally, the host and target computers will be one in the same. This simplifies software development and testing considerably. In many embedded applications, however, the host and target computers are separate. The host computer will normally have an editor and cross compiler along with some means of downloading the executable image to the target processor. The target system will typically have a monitor program running to allow for debugging of the application. Many cross compilers for embedded target processors are hosted on super minicomputers (like the VAX 11/780).

In contrast, many object oriented programming tools require expensive user interaction, windowing, and graphics capabilities; this is beyond the scope of most timesharing systems. As a result, these tools are designed for systems which can be dedicated to the individual programmer (like the Sun Workstation and the IBM PC/AT). Since these systems usually do not support cross compilation, the programmer must now design his program on one host, compile it on another, and debug it on a third. The extension of the software development process to three different systems introduces considerably more overhead; this can be especially taxing on a small programming team with a deadline to meet. In this situation tools which are supposed to increase software productivity may become more of a burden than a boon to development.

b. Portability

The importance of producing portable software is becoming increasingly recognized. As the cost of software increases, it becomes desirable to only write a piece of software which can be used on many different systems with little or no modification. This section provides insight into the why a tool should be portable and how this can be achieved.

(1) Rationale

It is critical that the intended scope of tool usage be defined before the design of a tool or the selection of an existing one for program development. There are three main levels of portability; these are:

-- Portable to a specific configuration.
-- Portable to a specific class of machines (such as the IBM PC,XT, and AT).
-- Portable to machines with different architectures.

Depending on the viewpoint, the need for portability will take on different levels of urgency. The tool developer may want to make his product as portable as possible in order to take advantage of a broader market (both existing and potential). On the other hand, he may decide to tailor his product for each system, thereby taking advantage of capabilities unique to that system. Although the applications programmer may not be initially concerned about tool portability, here are some reasons why portability may become significant:

-- The organization maintaining the software may not have the same host computer configuration as the original programmer. As a result, the software maintenance people will not have access to tools which served as an integral part of the design.

-- Routine replacement of equipment may make any tools purchased obsolete. This is especially important when one con-iders the lifetime of a weapons system compared to that of a computer system.

-- Upgrades to operating systems may also cause tool degredation.

(2) Portability Considerations

There are several factors which determine the potential portability of any given tool. These fall into three main categories; these are hardware, language, and operating system usage.

Perhaps the most important determinant of a tool's portability is how it interfaces with the hardware of its host. Truly portable code will have the following attributes:

-- It will be written in a high level language.
-- It will access peripherals through operating system shell calls, not directly
-- It shall not make use of special boards specific only to one configuration.
-- The amount of memory required to runthe application should not be beyond the addressing capability of any machine in a given class.

In some cases, the use of configuration specific code may be unavoidable. In this situation, all system dependent routines should be isolated into separate packages. These routines should be documented to include purpose, requirements, flowcharts; emphasis should be placed on details of the interface with the underlying hardware.

The programming language used to implement the design tool is also extremely important. Although using assembly language helps the tool designer to optimize its performance, it also minimizes portability and results in cryptic code. Using a high level language has the advantage of letting the compiler worry about potential system dependencies such as different types of addressing. The use of a high level language, however, does not absolutely guarantee portability. If the programming language is obscure, compiler availability on host systems may be lacking. It is also important to insure that variations in the language on different systems are minimized. Although compilers for a given language exist, there may be variations both in the compiler quality and the language itself across different host computers. A language which employs a standard (such as Ada) should help to guarantee some level of language integrity. Ideally, compiler vendors should support multiple architectures with the same front end to their compiler.

The choice of operating system also influences portability. The operating system provides services which allow the application program to interface to the underlying hardware. The flexibility of the operating system will determine if system dependent routines are necessary for tool development. If a programming tool is being developed soley for a specific configuration then system specific routines can be used with impunity. A good example of this is the window system available on the Symbolics 3600 series machines. The window system enables the programmer to create an array of customized windows and menus with relative ease. Unfortunately, however, there is no standard Lisp window system so the code produced is not portable between different implementations of Lisp.

The popularity of a given operating system is directly related to the portability of a tool which is built on top of it. If, for example, MS DOS is used, the tool should be portable between a given class of computer (like the IBM PC); or if Unix is employed, the tool may even be portable between different architectures. In addition, it is important that when calls are made to other programming shells that some standard exists for that shell. A good example is the use of the Graphics Kernel Standard (GKS) in order to produce graphics support for a portable application. The isolation of hardware dependencies, use of a standardized high level language, and the use of an operating system common to several architectures are three factors which must be used in combination in order to insure tool portability.

c. Usability

This section examines those aspects of a programming tool which influence the effectiveness of the environment as a whole.

(1) Overall System Considerations

A software engineering tool is only a subset of a total programming environment. Accordingly, the usefulness of any individual tool will be governed by the overall quality of its environment, despite its stand alone merit. In order to address this, the following criteria which should be considered when designing or selecting a particular tool.

The programming environment should be modular and extendible. The addition of new tools should not degrade the performance of the environment. Documentation should exist concerning interfaces between tools. In addition, system services (such as windowing and user interface routines) should be placed in common packages and made accessible to all tools in the environment.

The toolset in the environment should be functionally integrated. The IAW illustrates this concept by using diverse design tools to produce code for a common program. In this way the tools work as a set, not as stand alone components.

A centralized database should exist to share program information between tools. For example, the programmer may have the ability to review the PDL of the code he was modifying from any given design tool. All such tools should have access to the shared PDL database. These databases should be treated essentially as data structures. Operations upon these data structures should be formally defined. Additionally, these operations should be treated essentially as system calls by the requesting tool.

The environment should have a standardized user interface. The user interface should be easy to learn for the beginner (as are menu driven systems) yet it should not slow down the expert. A typical compromise is to make either a keyword or menu driven mode available to the programmer. This will let the user proceed at his own pace.

(2) Design Tools

The Interactive Ada Workstation prototypes gave AFWAL considerable insight into the the attributes of quality software engineering tools. The emphasis of the IAW was on tools to aid in program design and coding. The resulting lessons learned are summarized in the following paragraphs.

The design tools should integrate a number of different techniques. Each design technique has its strong and weak points when expressing different classes of problems. For example, a Buhr diagram editor is very good at expressing data flow, but does not touch on control flow. Additionally, a State Machine editor is an excellent way of expressing sequential control flow, but does not provide for concurrent communication between processes. Because each model has its own particular strong or weak point, it is important for the programmer to be able to exploit them appropriately.

The design tool should support automatic generation of source code. If each design tool is to represent a different methodology, it is important that the code generated by each be a code shell. This shell will represent the data flow or control flow algorithm described using the tool, but should have room to let the user fill in more specific details using a text editor. For example, a state in the State Machine editor could represent either a code block or a subprogram call. If a code block is represented, then the generated code should insert a placeholder which can be replaced with source code at a later time. In addition, each graphical design tool should operate off of a shared program database accessible to the other tools. As a result, the representation of the code will be consistent between different views of the program.

The design tools should have support for low level language constructs. With the advent of automatic code generation, the distinction between design and coding is becoming increasingly fuzzy. In addition, the programmer will often try to take advantage of specific language features (such as tasking in Ada) at the design level. While generic design tools will be composed of a subset of features shared by many languages, they will frustrate the programmer trying to write software in a specific language. As a result, the programmer will probably discard the design tool in favor of the familiar text editor. In addition, making the design tools language specific will also improve the quality of automatically generated code.

There should be a mechanism to expose inconsistencies in the design. A problem with many graphical design tools is that one can put a bad design in and get bad code out (garbage in, garbage out). On many representations, it would be wise to do several types of correctness checks such as:

-- *Parameter matching*
-- *Redundancy checks*
-- *Extraneous State checks*
-- *Design Simulation*

The generation of source code can be delayed if the programmer has some assurance that his graphical design is fundamentally valid. Once the code for a particular representation has been generated, the programmer will tend to edit the code and not the graphical representation.

(3) Coding Tools

The text editor continues to be the single most important software design tool available to the programmer. This is because programmers will tend to chose the tool which places the least inhibitions on their ability to design and code software. One fundamental problem with object oriented programming tools is that they often constrain the programmer to a few standardized methods of programming. This often results in the programmer discarding the tool when he finds enough operations which he cannot represent with it. Furthermore, if the programmer finds himself switching back between the text editor and another tool often enough, he will probably stick to the one he feels most comfortable with (most likely the text editor) in order to reduce overhead. Because of its importance, the text editor is one area where significant productivity gains can still be achieved; some guidelines to this end are outlined in the following paragraphs.

The text editor should support all standard editor functions. The text editor should support the insertion and deletion of characters, words, lines, and buffers. Buffers should be available to copy code into and out of; this is essential when similar code is used in different areas of the program. The editor should also have the ability to search for and substitute strings. Lastly, the editor should give its user the ability to move quickly to any part of the file. Standard editor functions should be invoked using keystroke sequences. The functions mentioned here are fairly basic.

The text editor should allow the user to edit two or more files at one time. Since many programs extend over several source files (with each source file being the equivalent of a package), this is a desirable feature to have. It may be desirable to implement this by partitioning the screen into multiple editor buffers.

The text editor's functionality should shorten the edit-compile-link-run-debug cycle. There are many ways to achieve this. One way is to evoke components of the programming environment such as the compiler and the interpreter can be directly from the editor. If semantic and logic errors are caught in the source file, modifications to the source code can be made and tested without having to continuously switch modes. In addition, if the editor is syntax sensitive, syntax errors be can caught and corrected as the code is being entered. This will reduce the number of "stupid" mistakes commonly made by programmers.

(4) Compilation System

Although the compilation system is the most important aspect of the programming environment, the functions performed by the compiler are being increasingly shared with other components of the environment. For example, the IAda editor in the IAW can perform incremental syntax and semantic analysis; these are functions commonly reserved for the compiler. The performance of a compilation system is the factor which will make or break the productivity of any programmer. A slow compiler will offset the advantage of any software engineering tool, no matter how useful. In light of this, it is important that guidelines for compiler quality be referenced in this report. The following performance guidelines for a compilation system are suggested:

(a) The compiler shall produce an object code program that requires no more than 30% additional target computer memory space over an equivalent program written in assembly language. [3]

(b) The compiler shall produce an object code program that requires no more than 15% additional execution time over an equivalent program written in assembly language. [3]

(c) The compiler shall compile a syntactically and semantically correct Ada program of at least 200 Ada source statements at a rate of at least 200 statements per minute (elapsed time), for each 1 MIPS of rated processing speed of the specified host computer, while meeting the above object code requirements. [3]

(5) Maintenance Tools

It is important that the software maintenance team have access to the same tools used by the developers. The use of object oriented programming tools will cause a great deal of design information to be embedded in the design representation. This is especially true if a PDL or requirements database is constructed relative to the graphical representations used by the tool. A switch to another environment may result in the loss of critical maintenance information, especially if the object oriented representations are treated as part of the software documentation. This is one reason why more emphasis should be placed on the portability of the programming environment. A programming environment which is relatively portable can be placed on the upgrades or replacement to a particular host computer (indeed this may need to be done several times over a given weapons lifecycle). AFWAL is currently working in-house on a tool called APEX (Avionics Programming EXpert) which will address the definition of a common software maintenance environment for avionics software.

IV. Conclusion

There are several trends which are significantly altering the way software is developed; these include:

-- *The increasing cost of software.*
-- *The increasing availability of cheaper, more powerful engineering workstations.*
-- *The rise of language and operating system standardization.*

These trends are resulting in the development of programming environments and tools which are both more portable and potentially more productive then their predecessors. Unfortunately, this new programming technology is still immature in many ways and potential users must be given guidelines to judge the overall effectiveness of these systems. These guidelines can also help alert either the tool designer or user to potential difficulties before committing to a potentially costly software development scheme. Because of the increasing cost of software development, the importance of evaluating programming environments is becomming more widely recognized. This paper presents a number of guidelines and considerations which can aid either the tool designer or applications programmer in judging a particular programming environment. These guidelines are based on lessons learned from a prototype software development environment (the Interactive Ada Workstation). By accomplishing this, this paper also gives insight into the direction that software development is taking.

References

1. Electronic Industries Associateion. "The Military Electronics Market: Perspectives on Future Opportunities". December 1985.

2. Nelson Weiderman; prepared for the Software Engineering Institute, Pittsburgh, PA. "Methodology for the Evaluation of Ada Environments". 5 Feburary 1986. SEI-86-MR-2. Page 8.

3. M.O. Hogan; prepared for the Space Division, USAF Systems Command, Los Angelos, CA. "The Definition of a Production Quality Ada Compiler". 18 September 1986. Aerospace Report TOR-008(6902-03)-1. Page 8.

RISE: Requirements and Incremental Specification Environment

Lt Kevin M. Benner, USAF
Command and Control Software Technology Division
Rome Air Development Center
Griffiss AFB, NY 13441-5700

ABSTRACT

An environment for capturing requirements and incrementally developing a formal specification is introduced. This environment, called RISE, is based upon Place/Transition (P/T) Nets. P/T nets are extended within RISE to support timing requirements and hierarchical decomposition as well as providing other extensions necessary for the development of a software specifications. The main focus of this effort was to define the semantics of hierarchical decomposition of P/T nets.

INTRODUCTION

In order to build successful software systems it is essential that all requirements be captured in the form of a specification and validated against user intent. This information includes functional decomposition, data flow descriptions, control flow descriptions, real-time requirements, and fail-safe/fail-soft requirements. Most specification systems currently in use do not provide the expressibility to capture all these requirements, nor do they provide an adequate means for validating the specification.

In [5] six criteria are put forward for the evaluation of a specification language/methodology. The criteria are:

a. Formality -- formal, unambiguous, mathematical basis

b. Constructibility -- build specification without undue difficulty

   (1) initial construction

   (2) captures programmers' concepts

c. Comprehensibility -- readability, size, and lucidity

d. Minimality -- does not over specify, focuses on "what" only

e. Wide Range of Applicability

f. Extensibility -- small change in concept results in small change in specification

Current specification systems address these criteria to varying degrees, often trading off strengths in some areas at the expense of others.

Petri nets, because of their formality and inherent ability to handle concurrency and synchronization, can be used as a specification methodology, but have proven inadequate when building systems of realistic size and complexity. In recent years, work has been done to provide hierarchical decomposition and an adequate representation of time. Recent work has also resulted in more expressive Petri nets than the original Condition/Event and Place/Transition nets [3], namely relation nets [4] and Predicate/Transition nets [2]. In total, this work has enabled a real world specification methodology based on Petri nets.

RISE, Requirements and Incremental Specification Environment, is based on the previously mentioned work. This paper will describe RISE in terms of its foundations in Petri net theory, its extensions to Petri net theory, and then describe future work based on lessons learned during this effort.

RISE FUNCTIONAL OVERVIEW

RISE breaks the specification process up into two main activities: 1) developing and/or evolving the specification and 2) validating the specification, at any stage during the development, against user intent. RISE provides the ability to specify, both graphically and textually, the system under development while allowing the user to incrementally develop the specification. RISE provides abstraction mechanisms which allow the user to elaborate arbitrarily the specification, putting off details until the specifier is ready to handle them. RISE allows for the specification to be partially validated at any point during the development. RISE does this by executing, or more accurately animating, its specification at the level the specification is captured. RISE does this by taking advantage of the net semantics which allow a net to be animated at the specification level without generating a low level implementation.

PLACE/TRANSITION NET FOUNDATIONS FOR RISE

This section will characterize place/transition (P/T) nets, describe extensions to support a hierarchy of P/T nets, timing requirements, and other extensions necessary for specifying software systems.

Definition 1: A P/T net is described as a 6-tuple, N = (S, T; F, K, M, W) in [3] where:

a. (S, T; F) is a finite net, the elements of S and T are called places and transitions, respectively. F defines the arcs which connect S outputs to T inputs, and vice versa;

b. K:S --> N, gives a (possibly unlimited) capacity for each place (where N is an integer);

c. W:F --> N\{0}, attaches a weight to each arc of the net (weight corresponds to the number of tokens which must travel along the arc);

d. M:S --> N is the initial marking representing the number of tokens in place s, respecting the capacities (i.e., M(s) <= K(s))

Definition 2: Firing rules for P/T nets: Under a given marking M(s) a transition $t \in T$ is M-enabled iff

$$\forall s \in {}^{\bullet}t : M(s) >= W(s,t) \wedge$$
$$\forall s \in t^{\bullet} : M(s) <= K(s) - W(t,s)$$

where ${}^{\bullet}t$ is the set of input places of t and $t^{\bullet}$ is the set of output places of t.

The follower marking is the result of an M-enabled transition firing. When it fires, places $s \in {}^{\bullet}t$ lose W (s,t) tokens and places $s \in t^{\bullet}$ gain W (t,s) tokens. The follower marking M' is described by:

$$M'(s) = \begin{cases} M(s) + W(t,s) \text{ iff } s \in t^{\bullet} \\ M(s) - W(s,t) \text{ iff } s \in {}^{\bullet}t \\ \text{otherwise } M(s) \end{cases}$$

If more than one transition is simultaneously enabled, only one will be arbitrarily selected and fired.

Graphically, places are represented as circles, transitions as rectangles, tokens as dots, and arcs as lines connecting S to T (and T to S). When transitions fire, tokens move through the net along arcs. This shows the dynamic behavior of the specified system.

EXTENSIONS TO PLACE/TRANSITION NETS

P/T nets as described thus far have clear, simple semantics with regard to capturing concurrency and synchronization, but have been inadequate when dealing with systems of realistic size and complexity. In particular, P/T nets do not provide a mechanism for hierarchical decomposition nor a mechanism to specify timing requirements (other than relative time relations, i.e., immediately before/after and arbitrary ordering).

RISE incorporates hierarchical decomposition and timed P/T nets similar to [1]. In summary, [1] used places to model system activities. This allowed transitions to retain their original convention of representing instantaneous events, while places on the other hand, represent activities which are active when a token is present within the place. An activity, which takes a finite amount of time, has a corresponding time argument associated with it. Therefore in a P/T net an activity is represented by a single place and has an associated time argument. Each place may be decomposed into a subnet which further elaborates the details of the associated activity and, in turn, each place of the subnet may be further decomposed. This allows for arbitrary decomposition of a specified system.

Time

RISE differs from [1] by using two time arguments (tmin and tmax) for each place rather than a single time argument. This allows the specifier to specify a variety of timing requirements on a place (e.g., a maximum time, a minimum time, a range, or an exact time), rather than a single exact time. Time arguments are used during net animation to check dynamically for timing requirements violations. Currently, static evaluation is not implemented.

During animation, time in RISE is updated like simulation systems with event clocks. When the next event occurs it updates the clock by the amount of time the activity took. This results in contiguous, non-regular time intervals which support arbitrary time fidelity. Within RISE each subnet has an evaluation mode argument used to specify best case or worst case timing (i.e., Did the activity take tmin or tmax time?). If the amount of time a token spends in the subnet does not satisfy the timing requirements of the parent place, RISE informs the user.

Hierarchical Decomposition

RISE supports two types of decomposition: open system decomposition (OSD), and closed system decomposition (CSD). Decomposition may be done on any place. The type of decomposition of the place depends on the type of elaboration the specifier desires. OSD is meant to provide lower level details of the parent place which are fully consistent with the parent place. CSD is also meant to provide lower level details of the parent place, but are not required to be fully consistent with the parent place. A CSD subnet is meant for refinement of the abstraction represented by the parent. A CSD subnet will remove generalizations assumed in the parent place and deal with exceptional cases not previously incorporated in the specification.

In both OSD and CSD, when a token enters a place with an associated subnet, it indicates that the net is in a certain state. The difference between the two types of subnets is best illustrated by how they act in relation to their parent place. A OSD subnet operates within the environment provided by its parent place. A CSD subnet replaces the parent place and provides a new, generally more specific environment.

Open System Decomposition -- Operationally, when a token enters a place with an associated OSD subnet, it is passed down the hierarchy to the subnet via an input transition. The token then passes through the subnet and is passed up the hierarchy via an output transition. The token being passed to and from an OSD subnet is similar to a function call with tokens analogous to arguments. The subnet simply provides a more detailed accounting of what the parent place represents.

OSD incorporates concepts from [1] for a precise methodology for decomposition. In [1] there are nine hierarchical net transformations, each of which may be applied iteratively to a simple place (i.e., a place with one input and one output) resulting in a more complex net. These transformations are: sequence of simple places, asynchronous parallel paths, synchronized parallel paths, predetermined two-way decision, data-dependent n-way decisions, predetermined two-way alternation, data-dependent n-way alternation, independent cycle, and bounded iteration. In RISE, a subnet must be provably derived via the decomposition rules previously listed, though for ease of use during specification development, this constraint is not constantly enforced, but rather is applied when directed by the user.

Closed System Decomposition -- Operationally, when a token enters a place with an associated CSD subnet, the place, with the token present, represents that the system is in a specific state. Information about the system must be derived from the subnet, rather than from the parent place. CSD subnets are closed Petri nets with initial markings. When a token is present in the parent place the subnet is activated. It will remain active until the token leaves the parent place. When this will happen is dictated by the net of which the parent place is a member.

With respect to the initial marking of the subnet, there are two types. In the first type of initial marking, tokens are always in the same place each time the subnet is activated. In the second type of initial marking, tokens are in the places they were last at when the subnet was last activated. Which type of initial marking is associated with the subnet is dependent on what the specifier states.

In a RISE specification, all nets are not necessarily related to each other in some hierarchical manner. Rather, the specification of most real systems would typically consist of multiple, essentially independent (i.e. interaction via shared resources only), simultaneous nets. RISE allows the specifier to describe such nets and animate these nets to see how they interact on shared resources.

Another deviation from [1], and applicable for both OSD and CSD subnets is the preservation of all levels of the hierarchy. In [1], decomposition is strictly one way. That is, when a place is further refined (decomposed), it is analyzed and animated only at this lower level. In RISE, all levels of the hierarchy are preserved, thus allowing the user to analyze the P/T net at whichever level of abstraction he/she wishes. (See EXAMPLE for more detail.)

Compound Places

In Petri nets, there is an implicit queueing of tokens in places. As a result there is some ambiguity with respect to what activities are being modeled. For example, consider the net below:

transition t1 can not fire because k of p2 would be violated.

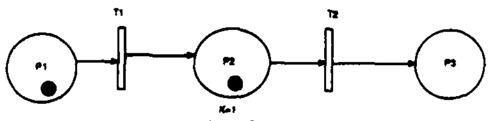


Fig. 1

What does the token in p1 mean? Is the activity associated with p1 still in progress or has it been completed and is simply waiting for t1 to fire. If there are any shared resources used by p1, when are they released? This queuing is necessary, but must be explicitly handled, so that a user specifies where and when queuing should occur. In RISE, this is handled by defining a RISE place as a compound place consisting of a sequence of three places: input queue (IQ), active queue (AQ), and output queue (OQ). The IQ is where tokens enter the RISE place and wait until they can enter the AQ. Tokens move from IQ to AQ when able. Tokens in the AQ indicate that the activity is actually taking place. When the activity is completed, tokens move to the OQ. Only tokens in the OQ can enable output transitions.

Two clarifications with respect to the time representation are necessary in light of compound places. First, the two time arguments refer to the time the tokens are in the AQ. And second, in addition to transition firing being instantaneous, so is the movement of tokens between queues within a compound place.

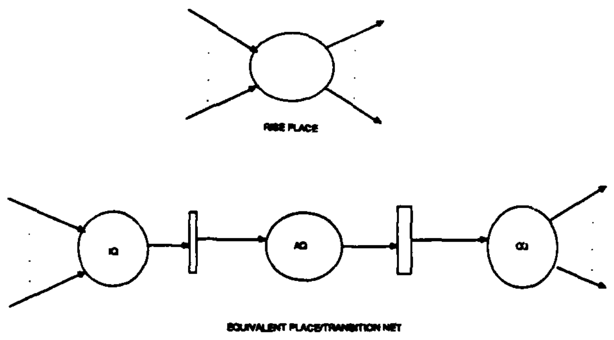


Fig. 2

Within RISE, all places are compound. As a result a single K bound (capacity) is no longer sufficient. Within RISE each place has two K bounds: K(place), capacity of the entire place -- sum of IQ, AQ, and OQ; and K(aq), capacity of AQ. These capacities put additional constraints on when tokens can move between queues, as well as, when a transition can fire.

With respect to hierarchical decomposition, subnets are associated with the AQ, rather than with the place as a whole. This works well for OSD subnets, but requires a modification of compound places in order to support CSD subnets. Places with CSD subnets contain only an IQ and an AQ. Once a token enters the

AQ it is eligible to enable an output transition and its activity will continue until the output transition fires (in contrast with standard compound places which complete their activity, move their token from AQ to OQ, and then enable output transitions from the OQ only).

EXAMPLE

In this section, a portion of an air traffic control system (ATCS) will be described. The purpose is to illustrate some of RISE's features.

We begin the specification by picking certain functions we wish to specify: tracking aircraft, running a monitor, and executing a hand off. Tracking aircraft (fig. 3) and running a monitor (fig. 5) are two continuous functions which must be performed as long as the ATCS is in use. Executing a hand off (fig. 4) is initiated by the air traffic controller whenever an aircraft leaves the air space of the controller and moves into the air space of another controller. At the top level these functions are essentially independent operations, hence at this level we have three separate nets. As we elaborate the specification and move down the hierarchy the interaction between these functions are specified.

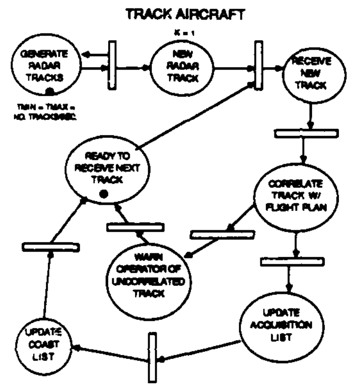

TRACK AIRCRAFT
K = 1
GENERATE RADAR TRACKS
TMIN = TMAX = NO. TRACKS/SEC
NEW RADAR TRACK
RECEIVE NEW TRACK
READY TO RECEIVE NEXT TRACK
CORRELATE TRACK W/ FLIGHT PLAN
WARN OPERATOR OF UNCORRELATED TRACK
UPDATE ACQUISITION LIST
UPDATE COAST LIST

Fig. 3

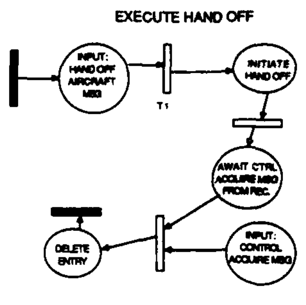

EXECUTE HAND OFF
INPUT: HAND OFF AIRCRAFT MSG
T1
INITIATE HAND OFF
AWAIT CTRL ACQUIRE MSG FROM REC.
INPUT: CONTROL ACQUIRE MSG
DELETE ENTRY

Fig. 4

Figure 3 is a net of how the ATCS handles radar track data. Each place represents an activity necessary to handle radar track data. For the purposes of this paper, the annotations within each place are sufficiently descriptive. In order for RISE to be used for realistic specifications, the annotations need to be formalized. As an example, places in both figure 3 and figure 5 could perform operation on the same elements of the database (i.e., the acquisition list and the coast list). This is not clear from the current specification. Part of this could be addressed within RISE by refining figure 5 down to include what makes up the database (e.g., explicitly state that the database contains the acquisition list and the coast list). This is clearer, but still ambiguous. This will be addressed in the future work section.

Currently in RISE, each place has seven attributes: tmin, tmax, tactual; number of tokens in IQ, AQ, and OQ; and a pointer to the associated subnet if it exists. To prevent cluttering of diagrams these attributes have generally not been provided in the example unless absolutely necessary.

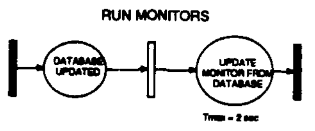


Fig. 5

Figure 4 describes the execution of a hand off of an aircraft from one controller to an other. Places "Initiate Hand Off" and "Delete Entry" have associated OSD subnets, figures 6 and 7 respectively. Place, "Await Control Acquire", is fully specified and will not be elaborated any further. Places labelled "Input: .. " are types of places which facilitate user interaction within the net.

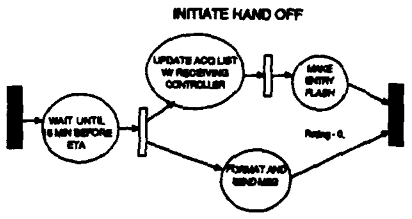

INITIATE HAND OFF
WAIT UNTIL 15 MIN BEFORE ETA
UPDATE ACQ LIST W/ RECEIVING CONTROLLER
MAKE ENTRY FLASH
FORMAT AND SEND MSG
Rating - 0.

Fig. 6

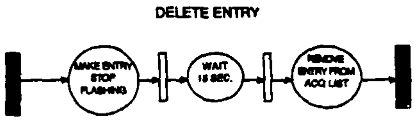


Fig. 7

Figure 5 is the top level specification of the run monitor function. At this level the only requirement we are specifying is that changes to the database be reflected on the screen within 2 seconds.

RISE provides the ability to execute the specification (animate the Petri net) at any level of detail. The user can select what level of detail or what portion of the specification is to be executed. This is accomplished by the user "activating" each subnet before execution begins. Once desired subnets are activated only those subnets will be used during execution. This facility can be used to present views of the system appropriate for the viewer. Management and top level system designers might only be presented the top few levels of detail. Domain experts might only see the portion of the specification relevant to their specific component while system integrators might see the entire specification or just critical components.

CURRENT STATUS

Everything described thus far is fully implemented. RISE is built in KEE 3.0 running on a Symbolics. The thrust of this effort has been on how to build and validate realistic formal specifications. The need for a graphical presentation was evident. Also necessary were mechanisms to support multiple levels of abstraction and incremental development, both of which are necessary for large specifications. It was felt that Petri nets had the necessary formality and simplicity to be a good graphical presentation. One of the biggest problems was to define the semantics of the *hierarchical decomposition* of places such that they would support multiple levels of abstraction and incremental development, hence the two types of decomposition: Open System Decomposition and Closed System Decomposition. At this point I believe there are other types of decompositions which would provide cleaner building blocks. The first step in this direction is to treat the type of decomposition, open systems verses closed systems, separately from whether or not the decomposition is an extension (i.e., more information which is consistent with previous information) verses a refinement (i.e., more information which may not be consistent and thus invalidate previous information). These issues are orthogonal to each other and should be dealt with separately.

FUTURE WORK

Future work will focus in two areas. The first area will be elevating the underlying Petri nets of RISE from Place/Transition nets to Predicate/Transition nets. This will allow constraints to be expressed as predicates inside transitions, while also allowing information to be carried inside tokens. Already completed in support of this work is a token description language (based on abstract data types) and the beginnings of a constraint/predicate language (based on first order logic and set-theoretic data types) The second area of focus will be on the interaction of refinement constraints and extension constraints during place decomposition. This work will identify the mechanisms necessary for elaboration and refinement of Petri nets through hierarchical decomposition. This work will draw heavily on work done by University of Southern California/Information Sciences Institute on incremental development of GIST specifications [6].

REFERENCES

[1] Coolahan, J., Jr., Roussopoulos, N., "A Timed Petri Net Methodology For Specifying Real-Time System Timing Requirements", Proc. International Workshop on Timed Petri Nets, Torino, Italy, July 1 - 3, 1985.

[2] Genrich, H., Lautenbach, K., "System Modeling with High-Level Petri Nets", Theoretical Computer Science, 13, (1981), 109 - 136.

[3] Reisig, W., Petri Nets: An Introduction, EATCS Monographs on Theoretical Computer Science, Vol 4, Springer-Verlag, 1985.

[4] Reisig, W., "Petri Nets with Individual Tokens", Theoretical Computer Science, 41, 1985, 185 - 213.

[5] Yeh, R., Current Trends in Programming Methodology, Vol 1, Software Specification and Design, Prentice-Hall, 1977, 7.

[6] Johnson, W. J., "Turning Ideas into Specifications", RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.

DISCUSSION

W.Mansel, Ge

Petri-Networks are an excellent method in modelling process control in real-time systems. However, they have the tendency to become complicated and difficult to handle and therefore are not easily accepted by software engineers. Can you please comment on the benefits of the RISE approach compared to the simpler methods of structured analysis with data flow and control flow control?

Author's Reply

The advantage of RISE over techniques like structured analysis lies with the greater degree of formality found in RISE Petri-Nets. With regard to the complexity of large Petri-Nets, RISE provides the necessary abstruction mechanisms to reduce this complexity.

The MBB Test Strategy and Tool Set for Software and System Integration

Dr. Hermann Hessel / Dr. Walter Wagner

Messerschmitt-Boelkow-Blohm GmbH
Unternehmensbereich Flugzeuge
Postfach 80 11 60
D-8000 Muenchen 80

Summary

Based on a generic system development concept a test strategy has been established within MBB Military Aircraft and Helicopter Division. In order to support this strategy a tool set has been developed which is being used for all projects and throughout all development phases. Due to its free configurability it can easily be adapted to production line applications as well as to other purposes.

1. Introduction

Over the last decade we have witnessed enhancing requirements of flying weapon systems and subsequently their growing complexity. However, this evolution is not fully described by the number of system components and their internal structures. With the introduction of bus systems into military aircraft all components merge to form just one complete system which is controlled by a continuously growing system software. The traditional simple separation of the system into software and hardware does not seem to be meaningful any more.Today the budgetary contribution of avionic systems to the total cost of a military aircraft is substantial. Hundreds of system engineers and software engineers are involved in the development of a flying weapon system. Besides the technical challenge this represents a problem of organization which we have solved by introducing a rigid methodology for system engineering.

Based on a generic system development concept (see paper no. 9 of this conference for details) a test strategy has been established within the MBB Military Aircraft and Helicopter Division that provides the appropriate facilities to support all phases of a system's life cycle. The focus of this paper is the description of these facilities, that enable the design, development, test and integration of complete systems. As it will become clear later, these facilities are not meant to replace the usual system and software development tools (like CORE/EPOS, PROMOD etc.), but are additional instruments.

2. The MBB Test Strategy

Figure 1 shows a simplified picture of a system life cycle. At all development and integration phases we provide facilities that support systems engineering. We will describe these facilities and their relevance to the different phases in this chapter in a generalized way and present the actual realization in the next chapter.

2.1 Tactic Simulator

In the very early development phase of a new aircraft the tactical requirements have to be worked out. Based on general military requirements the actual performance of the system has to be defined. This phase is supported by the MBB - UF simulation centre, offering various high fidelity real time simulations. Nearly all characteristic aircraft system parameters (speed, altitude, wing sweep, flap setting, etc.) can be modeled on the bases of approximately 70000 coefficients, (using an aircraft model in 6 degrees of freedom, aerodynamical rules etc.) and control laws can be worked out. Thus the performance of the parameterized aircraft can be tested under realistic conditions (cockpit with visual simulation) and

In principle two simulators can be linked in order to test combat performance. In this phase no real equipment besides some commercial displays and controls exist and the complete aircraft is simulated.

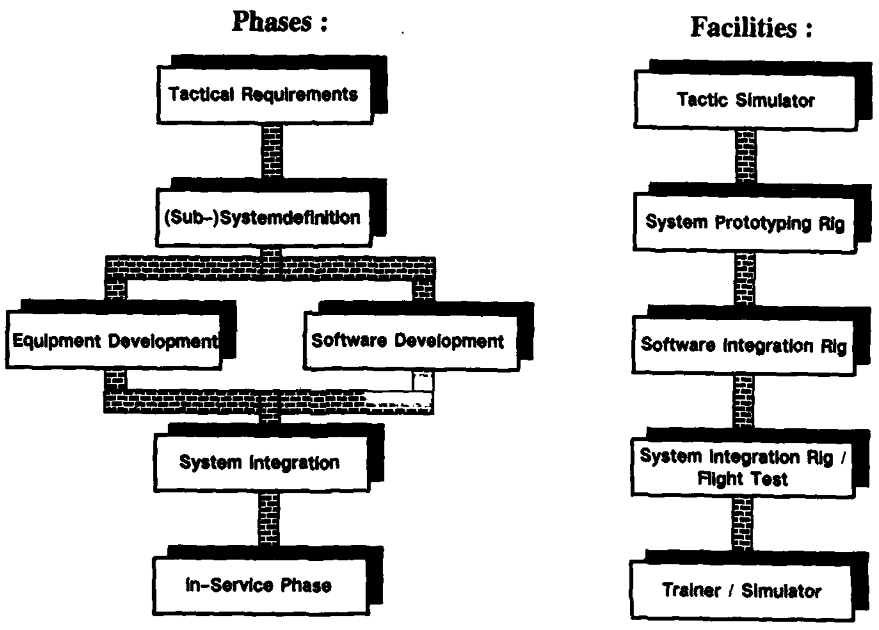


Figure 1 System Life Cycle

2.2 System Prototyping Rig

Once the tactical requirements are defined the system has to be designed. Experience in system development has shown that costs for error correction increase exponentially with the progress of system development.

Therefore MBB uses the concept of System Prototyping for early design verification thereby reducing risks and saving costs. System Prototyping means the experimental technical and operational assessment of system performance at an early development stage. It comprises investigations of system architectures and new technologies and allows to assess critical problems and ergonomic aspects not only theoretically but also by experiment. The System Prototyping Rig is a major tool for a disciplined and well controlled system development approach. The main functions of this rig are :

* System Analysis
* Tool Evaluation
* System Test Bed

System Analysis means experimental test and verification of new system architectures to support the system design. It includes the early verification and demonstration of system performance. Tool Evaluation means test and verification of software development environments. System Test Bed means the experimental tests of new equipment in a realistic system environment. It comprises in particular the pilot's assessment of man / machine interfaces in the cockpit e.g. of display formats, cockpit moding, arrangement of displays and controls.

2.3 Integration Rigs

Once the aircraft system is defined the equipment has to be developed and integrated. There is no canonical rule how to group equipment to subsystems and how many intermediate test stages are needed in going from the single module test to the final system integration. The guideline is to minimize the data traffic and the number of signal lines between the different groups, leading naturally to meaningful subsystems. Depending on the amount of work that has to be done in parallel the number of subsequent rigs has to be determined.

The test scenario that has proven to be efficient is described in Figure 2. Tests are performed at different levels of complexity, thus providing the necessary support at each stage to allow for parallel testing of subsystems and avoiding an overload at the final integration rig:

- Software Benches / Hardware Benches
- Subsystem / System Software Integration Rigs
- System Integration Rigs

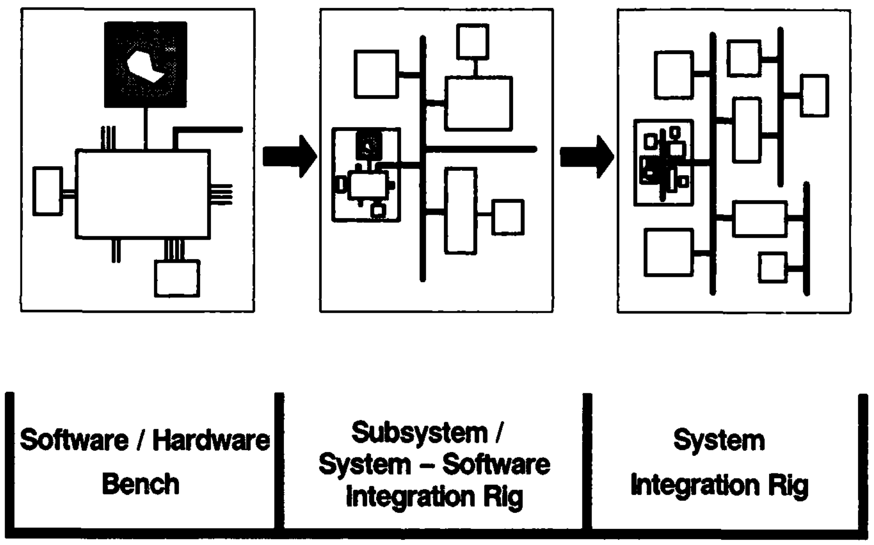

Software / Hardware Bench
Subsystem / System – Software Integration Rig
System Integration Rig

Figure 2 Stages of Testing

The test objects of the software benches and hardware benches are usually one piece of equipment, e.g. a dedicated computer or a single equipment. The benches are equipped with the basic corresponding displays and controls and facilities for example for loading the software into the target computer.Test objects of the Subsystem/System-Software Integration Rigs are the interplay of all the freely programmable on aircraft computers or subsystem equipment. Again, the peripheral aircraft hardware is limited to the basic displays and controls.The full system integration rig comprises all functional aircraft equipment with as much as possible of the original sensors in the loop and an almost fully equipped cockpit. This is the place where a test engineer can freely create his test scenario by configuring the system elements according to his demands.

2.4 Training Simulators

Derived from existing development and integration facilities MBB has provided several rigs for the in-service phase of a weapon system.For example a software maintenance facility for the mission relevant software maintenance or an avionic system trainer for the training of maintenance personnel.

2.5 The General Rig Concept

The general technical concept for the benches and rigs is such that all relevant signals of a test object are accessible via patch panels to simulation, stimulation and measurement equipment. The entire signal world of the test object is then accessible to interfacing equipment and a range of computer supported intelligence. The test engineer is supported by a user interface that has information about the details of the test object (like physical dimensions) through a 'test object description file'.

2.6 General Requirements to a Test System

On the one hand it is desirable that a test system is freely configurable according to the particular test requirements. Depending on the depth of the test different levels of sophistication are required for the equipment simulation. The amount of data that needs to be recorded and analyzed varies substantially. The required degree of test automation varies, depending on the repetition rate with which the test is performed resulting in a broad spectrum of realtime performance requirements to the test system.

On the other hand the main operations of the simulation, stimulation and measurement equipment are essentially identical throughout the different phases and programmes.

It should be evident to ask for a homogeneous family of test systems which can easily be adapted to the different test stages and tasks. The commercial market didn't solve our problem.

3. The Tool Set for Software and System Integration

In order to provide the appropriate test systems for each application MBB has developed a tool set from which a broad spectrum of test systems can be configured. This approach combines the advantages of dedicated single systems with the advantages of a big system that automatically guarantees commonality. Thus one saves costs by avoiding parallel development, training, education and maintenance of different test tools and still has all the required flexibility.

The name of this tool set, AIDASS, reflects its main functions: Advanced Integrated Data Acquisition and Stimulation System. It consists of three main components as shown in Figure 3.

- Multi Signal Interface Crate (MUSIC).
- real time test computer
- user interface with test object description

3.1 Multi Signal Interface Crate (MUSIC)

The coupling of all signals between the rig and the test computer is done via a standard interface device. It consists of a VME crate equipped with a CPU-board for preprocessing of signal information and VME interface boards for different signal types (e.g. analogues, discretes, synchros, MIL STD 1553 B).

3.2 Real Time Test Computer

Depending on the actual test requirements the data processing capability has tc be defined. The high performance end of the real time hardware spectrum is a multiprocessor computer which can be equipped with up to six parallel processing units. The test software is designed such that subsets can be created for lower real time requirements on less powerful computers, such as personal computers.

3.3 User Interface

In commercially available test tools, e.g. Milbus testers, a detailed knowledge about the internal signal representation (bit patterns) of the test object is required for the user to properly operate these tools. For integration purposes however the main focus of the test engineer is on the physical parameters and their logical interplay in the system.We have therefore developed an intelligent user interface that supports the test engineer by providing the detailed knowledge about the test object via a general test object description file.

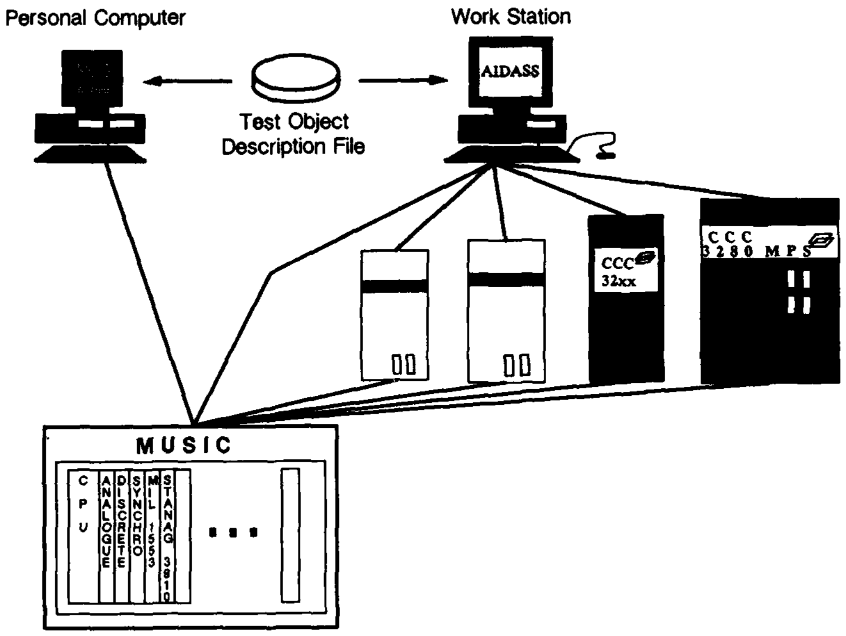


Figure 3. AIDASS Product Range

3.4 Present Status at MBB

At present we have developed both ends of the product range, the PC version and the full version implemented on a CCC 3280 MPS. As we have placed emphasis on software portability already in the design phase the implementation of AIDASS on a smaller computer can be done with relatively little effort.

4. Further Applications of the Test Tool Set

The concept for the test system AIDASS was primarily driven by the requirements for the development and integration of aircraft systems. Its application has been extended to flight test support and to the MBB production line. Presently we discuss with our partners in international programs to use this test concept in common to assure compatibility and exchangeability of development results.

Due to the fact that the AIDASS concept can be easily adapted to different test objects through a test object description file which does not affect the test software itself, AIDASS is not limited to the field of military aircraft development, but can be applied to other fields of military and civil development projects.

DISCUSSION

M.Muenier, France

Using your AIDASS system, are you able to "Replay" a test in its entirety (Regression Testing) or partially (i.e. till the point where an anomaly was detected) ?

Author's Reply

Yes, both can be done. Actually, in the full AIDASS configuration which I showed, this can be done in parallel to real time operation and test preparation.

E.Sevilla, Spain

How does the real avionics equipment fit into the AIDASS scheme? Are the discrete signals going from one avionic box to another interrupted for stimulating and recording purposes ?

Author's Reply

Not necessarily. For recording purposes the link is not interrupted but the signal is split between the test computer and the destination box. If equipment is simulated by the test computer we have two options. If the equipment can be deactivated electronically then the link can be kept alive, otherwise the link has to be cut and the simulated equipment is disconnected. This can be easily done on our patch panels.

C.Berggren, U.S.A.

Does your test computer communicate with the MUSIC box via Ethernet ?

Author's Reply

No. They are connected by a VME-Link.

"TOOLS FOR DEVELOPING REAL-TIME EMBEDDED SYSTEMS IN ADA - NEEDS, PORTABILITY AND EFFICIENCY"

Elisabeth Broe Christensen
Project Manager
DDC International A/S
Lundtoftevej 1C
DK-2800 Lyngby (Copenhagen)
Denmark

Summary

Ada offers some facilities, which were not part of older languages. If you want to take advantage of Ada for your real-time embedded applications, your development tools must support these facilites, and do it in an efficient manner. The paper presents the experiences from a tool project, the DDC-I symbolic debugger project, which has as its goal to provide a tool which can fulfil the demands of real-time embedded applications in Ada.

Introduction

The development of Ada was motivated by the problems of embedded real-time systems, like Avionics. Ada therefore - as an integral part of the language - offers a series of facilities which were not offered by the classical languages like Pascal and FORTRAN, such as parallel tasking, exception handling, representation specifications and inline-expansion. Furthermore, the language was standardized, in order to facilitate reuse and portability.

If you want to take advantage of Ada, development tools supporting all aspects of the language are needed. Furthermore, if you are developing embedded real-time applications, target program efficiency in terms of execution speed and storage requirements is important. To make such tools available to as many developers as possible at a minimum cost, the tools should be easily portable between different host computers, target computers and operating systems.

DDC International A/S (DDC-I) is a Danish company dedicated to providing efficient and advanced tools for Ada program development. Among other things, we are known for efficient cross-compilers and run-time systems. In such cross-environments you are both allowed to take advantage of the facilities of a general host computer for the program development, and to execute the target program in its natural surroundings.

However, in order to take full advantage of the cross-environment, you need an important tool in addition to the cross-compiler: A symbolic Ada cross-debugger. DDC-I therefore decided to expand its portable Ada compiler system to include a symbolic debugger as well. This debugger should

- support the features of Ada properly
- be useful for debugging of embedded real-time applications
- be easily portable

In the following it will be described how we have achieved these goals in the DDC-I symbolic debugger project, with examples of the problems we have found and the solutions we have chosen.

Proper Support of Ada Features

As mentioned above Ada offers, as an integral part of the language, some facilities, which were not offered by the classical languages like Pascal or FORTRAN. In order to allow you to take full advantage of Ada, your development tools - in this case the debugger - should support these aspects of the language. In the following some examples are given on Ada specific features that should be supported, and how they are handled by the DDC-I debugger.

The first facility to be thought of is tasking. The tasking model is an integrated part of Ada; an Ada debugger should therefore be able to handle programs with concurrent tasks. This means, that a strategy must be decided for handling the situation where several tasks are or may become breaked (i.e. suspended on a breakpoint); and the control structure of the debugger must be prepared for asynchronous events. Furthermore, the debugger should be able to handle the task states as defined by Ada. The current state of a task like "caller in rendezvous", "delayed" and "waiting for accept" and state information like current or expected rendezvous partner should be available, as well as information about the task structure like the names of the parent and the children tasks. Furthermore, the debugger should be able to set breakpoints at task events like task activation, rendezvous, termination, abortion and completion.

The DDC-I debugger supports debugging of Ada programs with tasks. A strategy and a series of commands for handling several simultaneously breaked tasks and allowing the user to inspect the different tasks have been designed. An extended visibility concept has been designed, allowing the inspection of the context of the different tasks as well as the value of objects declared inside package bodies. This extended visibility concept allows you to inspect the state of the entire system while debugging, instead of limiting you to the entities visible from the breakpoint on which your program (task) is currently suspended. This facility is particularly important in systems, where the state of one part of the system may affect the data produced by an other part of the system, as is the case in e.g. regulation systems with feedback.

As stated above, it should be possible - using the debugger - to display the entire task structure and the task states of the program and to set breakpoints at task events. Only, in order to do this, you need a unique identification for each task. As Ada allows the dynamic creation of tasks, the Ada name for a given task may not be designating a unique task, as several instances of that task may exist in the program at a given point of execution. The DDC-I debugger therefore assigns a unique task identification to each task in the system. The user may then use the Ada name when referencing tasks that are visible from the current viewpoint (point of execution of the currently breaked task, which is being inspected); tasks that are not visible may be referenced by means of the task identification. The visible tasks may of course also be referenced by their task identification if the user wishes to do so.

Another important feature of Ada is the **built-in checks** and the ability of well-controlled error handling - the **exception mechanism**. In Ada, it is checked dynamically that the executed code is legal; as an example, when a value is assigned to an object, it is checked whether this value is within the legal set of values (the constraints) for that object, and if not, an exception is raised. The user may define his or her own error conditions and associate a named exception to a given condition.

When the user during a debugging session wishes to change the value of an object in the target program, the DDC-I debugger performs the checks defined by Ada: If the assigned value is not legal, an error is reported. The DDC-I debugger also allows the user to set a breakpoint on the raise of a specific exception (or on all exceptions) in order to detect possible error conditions and investigate how they are handled. Furthermore, it is possible when using the debugger to explicitly raise an exception in the user program in order to test the handling of e.g. hardware failures, which may otherwise be difficult to simulate.

The last Ada feature to be used as an example here is the PRAGMA INLINE. This feature allows the programmer to structure a program neatly by defining subprograms without having to pay the efficiency penalty of a subprogram call. If a given subprogram is mentioned in a PRAGMA INLINE, the code of the subprogram is expanded at the place of the call; no call code is generated.

Most Ada debuggers do not support PRAGMA INLINE. Why is INLINE difficult from a symbolic debugging point of view? Basically, because the same source text statement or declaration correspond to several different entities of code.

The DDC-I debugger supports PRAGMA INLINE. A strategy and a set of commands for designating and handling a specific inline expanded call has been designed. It is worth noting, that the choice to support PRAGMA INLINE affects the entire mechanism for looking up names and deciding visibility; it is thus not a facility, which may be "added on later".

This concludes the description of how the first of the three debugger project goals: to support the features of Ada properly has been achieved. In the remaining part of the paper the achievement of the two remaining goals: to be useful for debugging embedded real-time applications and to be easily portable are treated.

Portability and Efficiency

Portability of a tool is important, because it allows the maximum number of users on various systems to profit from the tool with the minimum amount of time and effort. Furthermore, the learning overhead is minimized by using the same tool on different systems; and reuse, which is one of the key concepts of Ada, is facilitated by portability.

The Ada compiler system, of which the debugger referred to in this paper is a part, has been ported to a number of different development systems with hosts ranging from mainframes to mini computers and targets ranging from bare microprocessors to mainframes. Portability is therefore a key issue for this debugger project.

Easy portability of a debugger is not simple to achieve, as a debugger inherently manipulates machine dependent data.

The reason for treating the two goals: "Usefulness for embedded real-time applications" and "portability" together is that there is a potential conflict between the two goals. Portability in debuggers is often achieved by relaxing the demands for target program efficiency - in terms of speed and storage requirements - when debugging. Some debuggers rely on calls to special debugging routines to be inserted at all potential breakpoint positions in the original program code. In this way, the debugger is made portable as no knowledge of physical addresses is required by the debugger. The insertion of calls to debugging routines is called "instrumenting the code".

This approach is not acceptable for real-time embedded applications due to its effects on target program efficiency. The target program execution is slowed down considerably, causing the timing behaviour of the target system to change. Errors may change or even disappear when the code containing the debug calls is executed; or - even worse - critical system deadlines may be missed, causing other problems. Furthermore the storage requirements of the program are increased; in the extreme case, the needs of the debugger may cause the target program to grow to a size which makes it impossible to execute on the target system.

The debugger referred to in this paper is designed to require minimal interference with the target: No insertions in the target code are necessary. How is this achieved, and what is meant by still stating that the debugger is easily portable?

The requirement, that no insertions in the target code must be relied on, means that address information has to be handled on the host, and if the debugger is to be easily portable, preferably by the portable parts. This is achieved by defining a set of data structures (tables), containing the information necessary for translating between source text positions and physical adresses. The relations between the tables reflect the structure of the original source text.

The tables are built as follows. The portable parts of the compiler (the front end) generates the information about the program structure. The non-portable parts of the compiler (the back end) adds information about the relative addresses of the structures. Finally a portable tool builds the tables, adding the information about absolute addresses obtained from the linker.

When the user wants to set a breakpoint during a debugging session, these tables allow the portable parts of the debugger to translate the Ada source text position entered by the user to the address of the corresponding code on target. A breakpoint can then be introduced at that address on the target.

This approach means that no change or overhead in the target program execution occur, except at the breakpoints introduced by the user. Furthermore it means that all the information necessary for the debugger is stored on the host; no extensive target storage requirements are introduced by the debugger.

The target system interference required is then basically reduced to read and write operations on target memory and registers, and this is only done when the target program is suspended because a breakpoint has been encountered.

Some of the features of the debugger may though require changes in the run time system. One example is the facility of forcing a task to be suspended until explicitly released by the user: An additional task state value may have to be added.

In some applications this may not be acceptable. In order to allow for different user requirements depending on the application, and for different levels of system debugging support, the debugger is configurable: When implementing the debugger on a given host/target system, the implementor may choose to implement or leave out any facility. This is yet another aspect of making the debugger easily portable.

So, to summarize, what is meant by stating that the debugger is easily portable?

- the maximum amount of debugger processing is carried out by the portable parts
- the target (and host) interfaces are isolated and well-defined
- the data structures for information to be provided by target dependent parts are defined and prepared to be filled out, and a portable tool for building address information is provided
- the debugger is configurable, allowing for different applications and different levels of system debugging support

And what is meant by stating that the DDC-I debugger requires only the minimum target system interference:

- the debugger requires no instrumentation of the code
- the debugger may be configured such that no interaction with the target or additional overhead is required during target program execution between breakpoints
- the interactions with the target during program execution may be reduced to read/write operations on target memory
- the debug information is stored on the host
- all debugger processing of symbolic information is performed on the host

Conclusion

Some features, which tools for developing real-time embedded systems in Ada should have, were presented in the paper. Using the experience gained from the DDC-I symbolic debugger project it was demonstrated how a specific tool, a debugger, has been brought to possess these features.

DISCUSSION

W.Mansel, Ge

1. How is the host-target link realized for transferring debugging data from the target to the host? Is it done via RS232 serial link? Is high speed access via emulation possible?
2. What information can be gathered by task debugging? Are the task control blocks accessible?

Author's Reply

1. The debugger structure is designed to allow the use of any means available for host/target connection, including emulators. The first debugger targeting to be done by DDC-I (to the Intel 80 × 86 processor family) is to use a serial link (RS232 or similar), but it is likely that we will make an emulator-based version later.
2. When debugging tasks, information is available about the Ada task structure, i.e., the task ids, the current task states as defined in Ada, the task priorities as well as the current entry queues, caller in rendezvous etc. We have tried to provide all the information one could ever wish to have. Therefore it should normally not be necessary to look at the Task Control Block; but if you still want to look at the TCB, it is possible to use the machine level features of the debugger to display the contents of the TCB.

M.Muenier, Fr

1. How can you deal with acquiring (tracing) dynamically allocated data?
2. To what extent is your 80 × 86 system dependent on the RTS you developed?

Author's Reply

1. We use our knowledge about the compiler strategy for data layout and the information present in the symbol tables stored in the Program Library.
2. The 80 × 86 compiler is already being used with one externally manufactured Run-Time System and another one is planned. The Run-Time System Interface is described in a document and any Run-Time System adhering to this interface can be used. Therefore the compiler does not depend on the RTS.

   The debugger interpret RTS data structure parts depend on the RTS and would have to be rewritten if a different RTS is to be used. I would consider this a minor effort though, as the major parts of the target dependent parts of the debugger only rely on compiler generated information.

Three Generations of Software Engineering for Airborne Systems

A.O. WARD
MILITARY AIRCRAFT DIVISION
BRITISH AEROSPACE
WARTON
PR4 1AX
UNITED KINGDOM

SUMMARY

The serious practice of software engineering is a relatively recent phenomenon and there has been a minimum of reported experience in the use of methods and tools to support its application.

The technology can be viewed as progressing through several generations typified by the form in which development information is stored and accessed and the nature of the lifecycle. The former is described in terms of a file, data and a knowledge base, the latter as progressing from a phase to an object orientated lifecycle.

Experience with a typical first generation toolset is described briefly through the airborne software for the Experimental Aircraft Project. This is based largely on the use of structured methods for requirements and design, a programming support environment supporting the use of MASCOT and Pascal and rigorous management and control procedures. An advantageous cost benefit analysis of the project is reported.

A move to the second generation of software engineering is being planned under the auspices of the Eurofighter Project by the adoption of a Integrated Project Support Environment (IPSE).

The general characteristics of an IPSE are discussed including the tool interface and its relevance to emerging standardisation activities such as CAIS and PCTE Plus.

A longer term view of software is presented with an intimation of the scale and nature of future projects and a new form of lifecycle better suited to their needs. The requirements for software engineering to support artificial intelligence, safety critical applications and rapid prototyping are reviewed.

INTRODUCTION

In this paper we will be discussing three generations of Software Engineering but it would be useful to preface it with some simple, definitions and a down to earth assessment of the state of the art.

Boehm (1) makes a useful attempt, by first defining software and engineering individually and then synthesising an acceptable overall definition.

Software is the set of programs, procedures and related documentation associated with a system and especially a computer system.

Engineering is the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man in structures, machines, products, systems and processes.

Hence, software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful via computer programs, procedures and associated documentation.

However, an engineer in more traditional disciplines conceives a design to satisfy a requirement based on the use of modified or processed forms of raw material with known properties. There are catalogues of readily available components with stated qualities (both static and dynamic) from which to choose. Finally, there is an understanding of how they will fit and perform together which to a large extent is predictable.

This process is supported by many (sometimes hundreds) of years of science being applied to understanding the properties of materials and components, again both statically and dynamically. Mathematics has been a necessary component in developing this understanding and the whole is underwritten by a mass of test data.

Hence we may talk with some confidence about software management or order and list tools and methods whose functions have to a large extent been derived empirically but this does not constitute engineering in the traditional sense.

Taking this simple parallel with traditional engineering a stage further there is another fundamental difference between how the two communities approach their work. Today's designer would rarely consider it worthwhile to leave his drawing board for the workshop, roll up his sleeves and manufacture his design. Similarly the machinist who produces an individual component would not expect to be involved in fitting it into the overall structure.

Software designers, particularly in small organisations, function very much like the original and highly skilled engineers who set the pace in innovative engineering prior to the era of large scale production. They sketch an outline design, specify it in more detail, implement it at the coding level and then use individual modules to construct an overall system giving them responsibility for design, implementation and testing.

They are true artisans and it is not surprising that some of the tools that have been produced to help them reflect this view when they are referred to as software designer's or programmer's "workbenches". However, as systems have become larger there has been a gradual drift towards the methods of mass production and the segregation of the various tasks involved in production. A designer at a particular level need not be involved at the implementation stage and whoever is responsible for implementing particular modules would test them individually before passing them to a "systems integrator" to use them in system construction.

With the engineering of hardware the environments associated with design office, workshop and test house are vastly different and are tailored to meet the needs of staff involved in each of these activities. This reflects a fundamental understanding of the process that each environment is helping to support including the nature of the work and the skills and tools required to do the job.

We are many years away from the same level of understanding in software.

Progress, however, is visible as this paper attempts to show by describing the transition through successive levels of software engineering. This passage is witnessing a greater understanding of the development process and an increase in the degree of formality used for expression, both of which are imperative if the engineering of software is to approach its traditional counterpart.

THE FIRST GENERATION

In 1978 British Aerospace approved a small programme of research to examine ways in which the quality of avionic software requirements could be improved. A case had been made by pointing out the high leverage of good quality requirements on the subsequent implementation of software. This work was given further impetus by the growing realisation that if the airborne software for the next foreseeable project was to be implemented at productivity levels found on current aircraft projects it could exceed 1000 man years.

The initial programme focussed on system and software requirements but the scope of the project grew until a set of methods and tools emerged covering the complete lifecycle of software. The techniques were evaluated individually on a number of small projects but became fully integrated and used on a project of appropriate scale with the Experimental Aircraft Programme.

Collectively they represent a particular standard of software engineering based on a structured approach to requirements and design based predominantly on the use of diagrams. Language independent detailed design is linked to procedures for code production using different implementation languages. Software construction employs a programming support environment working on the host/target principle and support tools, in the main, are file based. This has been classified as a first generation of software engineering.

SAFRA

The overall approach is entitled Semi Automated Functional Requirements Analysis (SAFRA) and is based upon the widely accepted phase oriented lifecycle model. This is illustrated in Fig. (1) together with the methods and tools employed during each phase and these are described briefly below.

The technique used to develop and express requirements is COntrolled Requirements Expression (CORE   CORE provides the means to analyse and express requirements in a controlled and precise manner using mainly diagrams to represent data flow and data decomposition. The method comprises eleven logical steps which when applied to a subject requirement will decompose it into its lower level components. A CORE workstation (CWS) provides a means of constructing, editing and analysing diagrams for consistency and completeness.

CORE products may be subjected to more detailed analysis by use of PSL/PSA (Problem Statement Language/Problem Statement Analyser). PSL is an English like language, with a range of appropriate object types, capable of describing both architectural and behavioural aspects of systems. PSA provides a range of reports that will analyse the accuracy of PSL descriptions.

The interface between requirements and software design assumes the use of MASCOT (Modular Approach to Software Construction Operation and Test). MASCOT views a basic software design as a number of activities which operate independently and asynchronously but which co-operate by accessing shared Intercommunication Data Areas (IDA's) which can be either channels or pools. Integrating a requirements phase using CORE with a basic design phase using MASCOT is achieved by transforming a CORE requirement diagram into a MASCOT design.

Detailed design for activities involves normal structured program design, continuing to use CORE notation, and coding may be done in CORAL or Pascal. The software construction and testing phases make use of the PERSPECTIVE programming support environment produced by Systems Designers, PERSPECTIVE allows the incremental construction of software systems and their testing in a host machine and the target processor which is part of the avionic system. It also actively supports configuration management and so helps to ensure that valid versions of software components are used in system construction.

The Experimental Aircraft Programme

In May 1983, British Aerospace signed a contract with the UK Ministry of Defence to design and produce a new aircraft as a technology demonstrator, the programme to be known as the Experimental Aircraft Programme (EAP). The aims of the programme being to bring together a specific range of new and advanced technologies being developed by British Aerospace and other aerospace manufacturers in Europe.

EAP is a single-seat twin-engined aircraft, powered by two uprated versions of Tornado's Turbo-Union RB-199 engines. It has all moving foreplanes (canards) and an advanced compound swept wing. It is designed to prove in a one-off demonstrator the latest technologies which have been the subject of separate government research contracts over the past decade. The various technologies being tested and integrated into this demonstrator are utilised in the four nation European Fighter Aircraft (EFA) project currently being developed.

Some 50% of the UK costs have been provided by HM Government as its contribution to the design and building of this aircraft. The balance has been found by British Aerospace and its UK and European partners.

The main areas of advanced technology are to be found in the active flight control system, the avionics suite, the extensive use of carbon fibre composites, aluminium lithium, and in the use of super-plastic forming and diffusion bonding techniques in the airframe construction.

The EAP system architecture comprises three major sub-systems:-

- Flight Control System (FCS)
- Utilities Systems Management System (USMS)
- Avionics

Communications between the three sub-systems is achieved via Remote Terminals on the main Avionics data highway. The FCS was a derivative of the existing Jaguar Fly-by-Wire demonstrator aircraft system. However the USMS and Avionics systems represent new design work and SAFRA was specified as the toolset to be used for the specification and design of these sub-systems.

The USM System is responsible for software control of many of the mechanical aircraft systems that have traditionally been controlled by analogue means and examples include Hydraulic control, Fuel Management and Braking systems. The Avionics system provides the processing of raw sensor data, overall system and displays/controls moding and the generation of a wide range of multifunction cockpit displays. It also has executive control of the total systems, including bus transactions and remedial actions in the event of failures.

Preliminary Results from EAP

Comparisons have been made between producing the software for EAP and avionic software from previous projects, looking first at the gains.

- Errors are detected sooner; early experience with Tornado had prompted the use of more dynamic testing techniques to clear the Air Defence Variant software on the rig. This caused a significant shift from the aircraft to the rig in detecting problems. The evidence on EAP shows that problems were being detected much earlier in the lifecycle.

- Quality of delivered software is improved; comparing EAP with the most previous example of avionic software to go through flight clearance testing on the rig shows that quality has improved by almost a factor of ten.

- Less resources are needed for rig testing; improvements in quality have led to reductions in the volume of testing. Test schedules are still exhaustive but the amount of retesting as a consequence of change is now much less.

- Productivity is higher; the two most recent projects were experiencing individual productivity of about 250 lines of code per man year for software design, code and test. The average productivity achieved on EAP was 1400 lines of code per man year.

Less quantifiable gains are typified in the approach providing a detailed framework for training (rather than learning on the job). This meant that a comprehensive and detailed training scheme could be planned which in turn led to teams being formed reasonably quickly from relatively inexperienced personnel. It also demonstrated that putting more people on software projects can make then earlier, albeit at the expense of individual productivity.

The introduction of the approach did encounter some problems:

- There was some preliminary resistance to the use of SAFRA which, although it was overcome as benefits emerged, could have been reduced by better education in the value and role of SAFRA.

- Some of the tools were immature demonstrating the limitations of using small evaluation projects to fully prove a technique. Availability did not necessarily mean maturity.

- New tools, quite naturally, will still be evolving and this can potentially have a disastrous effect in a project context. Proposed developments must be identified and special arrangements made to freeze tools if necessary.

SAFRA corresponds to a first generation of Software Engineering technology. The tools are predominantly file based and integration is functional rather than internal. Some limitations with this standard include:-

- the problem of embodying additional or different tools
- the means of automating the enforcement of configuration control and management procedures over the whole lifecycle.

These limitations will be lifted with the advent of a 2nd generation of software engineering typified by the use of Integrated Project Support Environments (IPSE).

THE SECOND GENERATION

The loose coupling of tools typified by the First Generation, relies on zealous application of manual configuration management and control procedures to handle the transposition from the file store of one tool to the next. While this is practicable in the context of a relatively small project, such as EAP, it is inadequate for a large international project of the scale of Eurofighter.

This project is likely to be at least an order of magnitude greater than EAP, the implementation will be much more widely distributed and development will take place over a longer period of time. It can be argued that the nature of methods and tools being used will remain about the same.

Structured methods will be used for requirements analysis and detailed software design, while basic design will be supported by a high degree of modularisation that will map on to the higher order language, Ada. The latter will use a compiler that allows host/target testing.

However, the volume of design information produced over the complete lifecycle of the Eurofighter implementation demands much more than a loose coupling of tools through individual files, it requires a single repository of information in the form of a centralised database.

As Tim Lyons argues in (2) the essential part of any support environment is the ability to store data. Projects typified by Eurofighter will have an enormous amount of data, embracing requirements, specifications and design documents, test schedules, source code, object code, executable binary images and management and control information. to date this has resided in the individual files of tools appropriate to each function.

However, the information that a project needs to store consists of not only these individual entities but also the relationships between them. These relationships include those between source and object file, design and specification and the connection between a change and a error report.

The notion of a database as the centre of an environment is to provide explicit support for representing any of the data normally held in a file, but also the relationships between the data. The database provides support not only for representing data items but for representing the structure of data.

Clearly the existence of such a mechanism provides the basis for rationalising the interface between the database and the tools that will make use of it. This interface is popularly referred to as the Public Tool Interface (PTI) for an IPSE and is shown schematically in Figure 2. The Figure goes further in that it illustrates the other skins around the database that may or may not be an integral part of the PTI. These include the version and variant control aspects and the rationale of how the user interacts with tools. An example of the latter is standardising on a particular style for editing or the provision of help information.

The model also accepts the notion of the unpopulated IPSE being used on different projects with appropriate tools and those specific to a particular organisation or company. This includes tools developed independently of the IPSE (3rd party tools) and those that will not interface through the intimate connection of the PTI. The latter will occur when tools already exist, are not required as intensively throughout the project and the cost or technical difficulty of such an interface is beyond the scope of the user. This latter foreign tool interface is more likely to be at a file rather than at the detailed object level.

While a growing number of proprietary IPSE's with tool interfaces of this kind are beginning to emerge there is also considerable interest in PTI's as a standardisation activity. Such a move is unique (and somewhat adventurous) against a background of very little experience in the use of IPSE's. However, work on both sides of the Atlantic is pursuing the objective of a standard for particular PTI's through the CAIS and PCTE programmes of work.

In 1982 the US Ada Joint Programme Office established a joint service team known as the KAPSE Interface Team (KIT) and a complementary industry/academia team (KITIA). The object of the team's work was to co-operate and converge on a set of Ada Programming Support Environment (APSE) interface standards to permit the sharing of tools and other software between DOD supported APSEs. KIT and KITIA have developed a Military Standard Common Apse Interface Set (CAIS) and in parallel have also developed the requirements for a more advanced interface set which is being developed by a US Industrial Consortium.

In Europe, a consortium under ESPRIT are developing a system as a basis for a Portable Common Tool Environment (PCTE). The PTI of this system is intended as a common platform on which tools can be built, both within the ESPRIT programme and elsewhere. PCTE continues to be developed as part of a further European programme (under IEPG/TA13) towards an enhanced civil as well as military standard. This is due to enter the implementation and assessment phase in the near future. It is hoped that there could be conversions of the US and European standards, possibly under the auspices of NATO.

THE THIRD GENERATION

Aerospace software projects continue to represent some of the most challenging examples of large real-time embedded systems. Projects with lifecycles in excess of 15 years are not uncommon and, even with the highest productivity currently available, implementations can number several hundred man years.

Airborne applications in the last two decades have reached several hundred thousand lines of code, during the early 90's systems are more likely to contain several million lines of code. In order to contain the human resources required we seek productivity improvement to match this growth.

An order of magnitude improvement over current projects is the eventual target.

The need is compounded by other industrial sectors experiencing a similar growth of software in their products and competing for the services of the same people required by Aerospace.

A similar growth is to be found in the scale of safety critical systems as shown in Figure 3, illustrating that improvements in productivity must not be at the expense of quality. The major advance between the lower points and EFA are the likely move towards the use of a safe sub-set of a higher order language and formally based code analysis tools. A further advance is required in order to meet the scale of safety critical systems as typified by the HOTOL Spacecraft (Horizontal Take Off and Landing).

The issue of scale is a special one as it leads to speculation as to the main drivers that will lead to further reductions in lifecycle cost. BAe's experience with mission (as opposed to safety) critical software on EAP has led us to conclude that further significant reductions in lifecycle cost will not result from pursuing improvements in quality.

The SAFRA approach, which concentrates effort on the earliest stages of the lifecycle has so reduced the scale of testing required that the latter is a relatively small percentage of the overall lifecycle costs. If we seek an order of magnitude improvement in overall productivity, this must come from physically reducing the scale of the implementation task. This will only emerge through large scale re-use of software, either by employing software components or specialised high level languages.

The other major driver on future software engineering requirements lies in the nature of avionic systems themselves. Until recently, the typical airborne system was largely self contained, reflecting the independent nature of the operational requirement. The introduction of datalinks, airborne command and control stations and the concept of overall battle management as typified by Figure 4, is likely to change this view. This broader picture will effect the nature of airborne system requirements in two ways:

- Systems will be in a perpetual state of evolution
- They must be considered and developed with a view to interfacing with a wide range of other systems.

This change in the nature of systems is likely to affect the basic lifecycle model that underlies current implementations.

The typical model (as shown in Figure 1) is popularly known as the "Waterfall" and has each successive phase being defined in terms of input, output, method to generate the output and the tools used to support the application of the method and validation of the output. Iteration takes place between adjacent phases and the whole lifecycle can be repeated to develop successive releases of software over the life of a system. The model, however, rests on the assumption that it is possible to define the requirements completely before proceeding with implementation and design. An object orientated approach to systems is a recognition that the completed system will only be arrived at through a succession of refinements.

In other words, the lifecycle is geared to establishing a better understanding of the objective of the system.

With such an approach the need for two logically parallel lifecycles emerges, addressing prototyping and system development respectively. The lifecycles co-exist and are served by methods and tools appropriate to them. It is likely, however, that although there is a logical separation of activities and phases they could be supported by a physically integrated set of tools.

The approach is categorised by Figure 5. The detailed nature of interaction between the lifecycles is a function of the type of prototyping which can typically fall into the following categories:

- Exploratory
- Experimental
- Performance
- Organisational
- Evolutionary

The predominant concern of the prototyping lifecycle is to understand what the eventual product will be. This, in most cases, could be at the expense of real world performance, resilience or integrity except where the nature of the prototype is to address those issues.

With the system development lifecycle the importance of producing well engineered software still applies and all the relevant features of the current lifecycle model will be retained.

The challenge of both scale and nature will be met by a new generation of software engineering and an equivalent project support environment as typified in Table 1. Just to recap for a moment:

- The first generation made use of design techniques and tools over the whole lifecycle of software;
- The second generation is destined to provide a common infra-structure that supports the tools and hence a full software design and management environment.

The third generation will extend this environment to a new lifecycle model and development techniques and tools to improve productivity by several factors. One of the key enabling technologies will be the use of artificial intelligence and hence the presence of a knowledge as well as a database at the heart of the support environment.

The construction of such an environment is not trivial and represents a considerable advance over current programmes of research and development but is likely to build directly on their results. It is not a task that can be undertaken by a single company and it is for this reason that five major European Aerospace companies have proposed that they collaborate on a third generation environment entitled AIMS (Aerospace Intelligent Management & Development Tool for Embedded Systems) under the auspices of the EUREKA programme. The preliminary thoughts on AIMS are offered as being typical of a third generation environment.

The Scope and Nature of AIMS

The role of AIMS

AIMS will provide a development environment for systems or subsystems containing embedded software components. It must do so in a context of co-operative work between teams from several European countries and cover all stages of development for those parts of the system dependent on software components.

We will first discuss the major target areas of system development that AIMS will address before describing some technical aspects of the AIMS environment.

Major Targets

- Management Support

  An integrated environment for co-operative international work will be expected to provide extensive facilities for project management. Typically this should include:-

  - development planning with day to day assessment of achievement
  - control over development schedule and cost
  - quality control and management of the cost of design evolution and modification
  - configuration management

- Development Organisation

  International collaboration causes teams of developers from various cultures or backgrounds to work together over many years. Communication and flexibility are important in the environment they will be using.

  - Methodology-independent communication tools for data and knowledge exchange will be essential so that teams from different companies may continue to use their own sets of design methods and tools. They must, however, be able to communicate freely all kinds of design data in an homogenous way.
  - A homogenous environment must be able to adapt to new tools and the design methods they support, as well as being able to integrate advances in man-machine interface or workstation technology.
  - Acquisition, storage and restitution of system-specific and engineering knowledge and experience should be carried out at automatically as possible.
  - Support for systems with a long life must be provided, including the storage of knowledge as a means of improving overall system design consistency and maintaining it over time.
  - Design methodologies evolve with time and functions like documentation, either for the user, designer or configuration manager, must be supported in such a way, as to be independent of the particular design methodology used.

- Productivity

  AIMS will have to support the development of large, modular, multi-technology applications, with multiple teams that can use different sets of tools and design methods.

Reducing development and maintenance costs involves:

- reducing the number of iterations from requirements to implementation and delivery of the system.
- providing expert assistance at each development stage for efficient tool usage, overall consistency checking and re-use of previous developments.
- reducing the impact of corrections and modifications and improving their reliability. This can be achieved from preparation in the conceptual stages, preserving consistency of design options over time, keeping track of the decisions made and of the reasons behind them, and using that knowledge for information and training of newcomers to the project.

• Communication

There are two facets to this subject:

- The "semantic" communication between human decision-makers and involving elaborate mailing, expert (knowledge based) assistance, and thorough co-ordination with project management.
- The exchange of technical data through heterogeneous computer networks and different tools. System descriptions must be accessible through formal means of expression, independently of the tools used, so that they can be transferred from one tool to another without the need for knowledge of a specific tool in order to be analysed.

  There is of course the problem of confidentiality with industrial data, and the access conditions to information must be well controlled.

• Expertise

Knowledge-based assistance must be provided for many reasons:

- Helping specialists to co-operate and communicate across distance (European projects) and time (product updates). A knowledge-based simulation of specialists can help designers when faced with technological choices or assessing the implications of the given technology. It can complement and accelerate the necessary dialogue between specialists, chronology of decisions and how they were reached. Re-use of this knowledge as expert assistance will be provided for long-term design maintenance; it is also the most flexible means to support re-use of previous developments.
- Domain-specific as well as tool specific knowledge are hard for individuals to acquire. If it is available in the development tool it will help designers to improve their capability, and avoid redundant and duplicated origins.

• Simulation

Over the lifecycle of such complex and reliable products as aerospace systems, every step must be validated. This holds especially true during the early definition stages, when all decisions potentially have a large impact on development cost (feasibility) and exploitation cost (quality).

It is necessary to provide the design with tools for extensive validation and evaluation of these decisions with high financial leverage through the use of effective simulation and prototyping tools.

A secondary but important practical consequence of prototyping is that it acts as a basis for the generation of test and integration procedures for later stages when the systems are integrated.

• Tool Support

AIMS is expected to have the capacity of integrating and supporting new CAD tools as they become available. This support includes data format conversion to a common standard, documentation and expert advisory functions.

AIMS will have to cope with very sophisticated tools mixing advanced computing techniques (networks, data and knowledge bases, expert systems, graphic man-machine interface, etc) in very heterogeneous environments. The implementation of AIMS must provide as much portability as possible.

The AIMS Environment

The essence of the infrastructure consists of a database surrounded by shells committed to configuration control, tool interfaces and MMI, being typical of current Integrated Project Support Environment (IPSE's) as described above.

In moving to an environment that will support the AIMS concept there is a growth in the number of functions surrounding the database and the added requirement of storing and accessing a knowledge base.

- Proposed Model for an AIMS Environment

The model is illustrated in Figure (6) and can be said to consist of:

- A data and knowledge management kernel
- A tool support framework
- Man-machine interface

A data and knowledge management kernel

- An object orientated data management system to store project representation (working prototypes with documentation and development data and interface drivers to simulators and test benches). Object support should include message sending, pattern matching and data retrieval. While project support requires a version and variant support utility.
- A set of knowledge bases with their consultation and updating facilities, general inference mechanism, dialogue support and archiving. These knowledge bases would cover:
- General Knowledge (mathematics. physics)
- Domain specific knowledge (navigation equipment, customer needs)
- Design expertise (methodology, known solutions)
- Product-specific expertise (weak-points design history)

Most data and knowledge is of course proprietary to the company or even the team using AIMS so its availability to other users (partners, subcontractors) will need to be carefully controlled.

- A tool support framework

Tools must continue to be interfaced to the kernel and to one another. The framework must provide a homogeneous environment for the tools to use taking into account the following:

- User information and advice, for a consistent style of working across projects. This includes methodological help, support for knowledge base consultation and user to user communication. It must not be forgotten that typical system development organisations will be very "distributed" so, as mailing is a very "smart" application, it must be complemented by expert system consultation to speed things up.
- Support for all kinds of design tools with a convenient interface structure to the kernel to other tools and to the user. An outer layer would provide user guidance for proper use of the tools.
- Special support for project manager(s) and lifecycle issues such as quality management and configuration control.
- Special support for Host/Target testing particularly at system integration, including provision for fault diagnosis.

AIMS is about to enter the definition phase at the beginning of a five year programme of development and implementation. The collaborating partners include:

- Aerospatiale
- Aeritalia
- British Aerospace
- CASA
- MBB

Software houses in each individual country are also likely to be involved where specialist expertise is required.

It is hoped that as well as providing an advanced environment to meet the needs of the partners, AIMS will also provide some of the framework for European standardisation on future Software Engineering issues.

ACKNOWLEDGEMENTS

I would like to acknowledge the immeasurable contribution of my colleagues to the content of this paper and to thank British Aerospace for their permission to present it.

I would also like to acknowledge the support of the Procurement Executive of the Ministry of Defence.

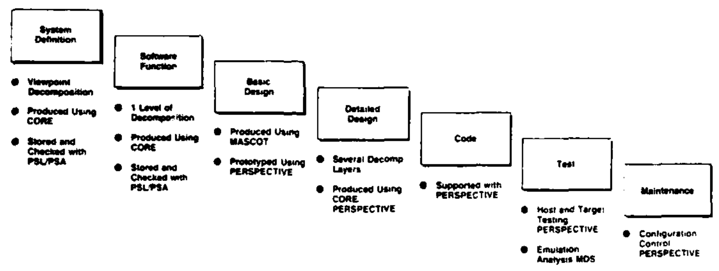
REFERENCES

1. Software Engineering Economics B.W. Boehm 1981

2. The Public Tool Interface in Software Engineering Environments T.G. Lyons IEE Software Engineering Journal Nov 1986

BRITISH AEROSPACE

Fig 1. First Generation . . . The Solution

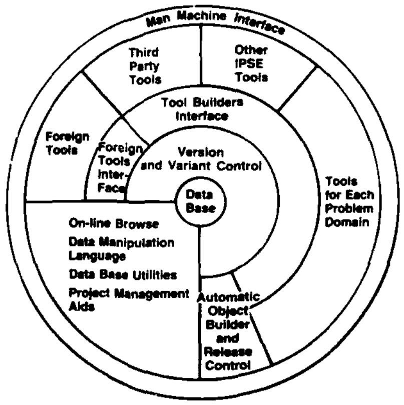

System Definition
● Viewpoint Decomposition
● Produced Using CORE
● Stored and Checked with PSL/PSA
Software Function
● 1 Level of Decomposition
● Produced Using CORE
● Stored and Checked with PSL/PSA
Basic Design
● Produced Using MASCOT
● Prototyped Using PERSPECTIVE
Detailed Design
● Several Decomp Layers
● Produced Using CORE PERSPECTIVE
Code
● Supported with PERSPECTIVE
Test
● Host and Target Testing PERSPECTIVE
● Emulation Analysis MDS
Maintenance
● Configuration Control PERSPECTIVE

. . . A First Generation of Software Engineering

BRITISH AEROSPACE

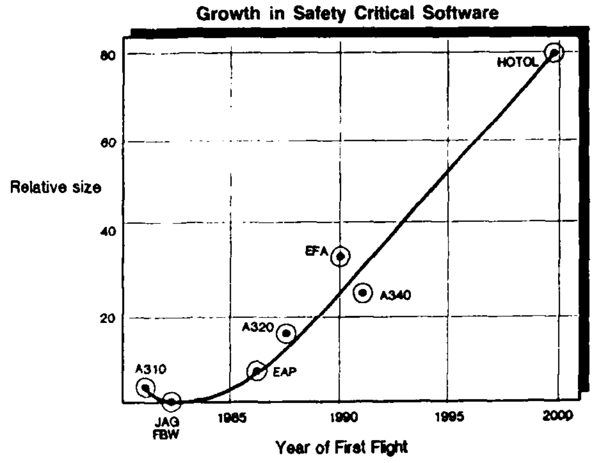
Fig 2. Second Generation: The Solution

The Integrated Project Support Environment



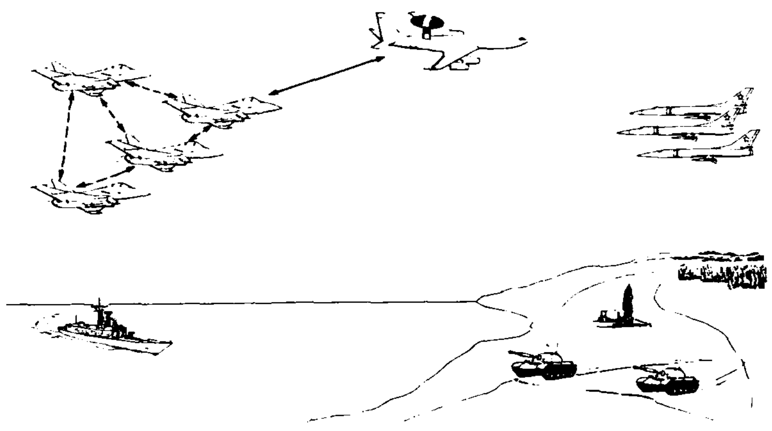
BRITISH AEROSPACE

Fig 3. Third Generation: The Problem



BRITISH AEROSPACE

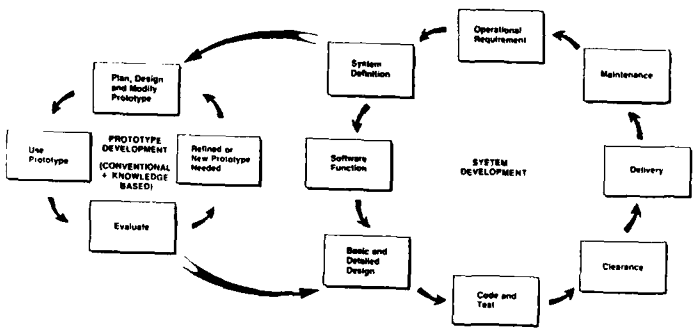
Fig 4. The Nature of Systems is Changing



BRITISH AEROSPACE

Fig 5. Third Generation: The Solution

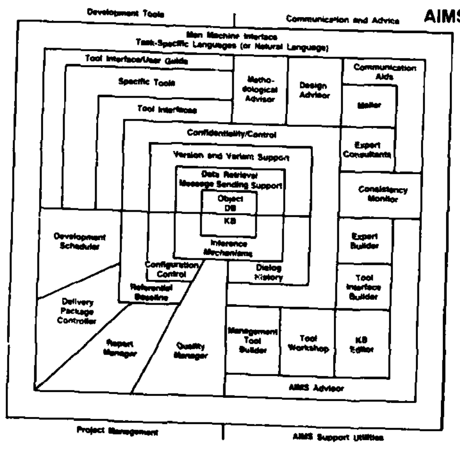
A New Lifecycle Model


Operational Requirement
System Definition
Maintenance
Plan, Design and Modify Prototype
Use Prototype
PROTOTYPE DEVELOPMENT (CONVENTIONAL + KNOWLEDGE BASED)
Refined or New Prototype Needed
Software Function
SYSTEM DEVELOPMENT
Delivery
Evaluate
Basic and Detailed Design
Clearance
Code and Test

BRITISH AEROSPACE

Fig 6. Third Generation: The Solution

AIMS Environment



SOFTWARE READINESS PLANNING

Jack Cooper
Anchor Software Management
5109 Leesburg Pike, Suite 214
Falls Church, VA 22041

Summary

This paper presents a method wherein the capability to support and make changes to avionics system software during the readiness phase of its life cycle can be guaranteed. In recognizing that the supportability of software is completely determined during its development, the method centers around pre-development planning, decisions and actions on the part of the avionics system's acquisition manager. Discussed are the software engineering considerations that must be included in the contract to facilitate the future supportability of the software. That is followed by a description of contractual requirements to be placed on the software support environment used to develop the avionics software to ensure the life cycle supportability of the target avionics software. Two tools, a management Plan and a contracting Standard, that support the method for ensuring software supportability are described. The Plan provides a vehicle for joint, customer and contractor, management of the developmental software support environment. The Standard provides the contractual life cycle supportability requirements.

I. Introduction

The magnitude of all operational and support software for an avionics system could amount to millions of lines of code. The development and life cycle support of this quantity of software is a major project consideration. Consequently, it is very important that planning for all aspects of these efforts be completed <u>before</u> software development begins. A Software Life Cycle Management Plan is an <u>excellent</u> vehicle for this planning. Such a document should be completed prior to entering into a contract or internal task agreement. A Software Life Cycle Management Plan will preclude any of the important aspects of software development and support from becoming a fait accompli.

Since planning for a software development project has been widely discussed elsewhere, this paper focuses on planning issues related to supporting software during the readiness phase of its life cycle. It is paramount that software life cycle readiness issues be provided for in the initial software development contract or task agreement. Current estimates are that something in excess of 75% of the total cost of the software is expended during its readiness phase. Since the requirements for developing and supporting software are not always the same, all acquisition tradeoffs should be weighted 3-to-1 in favor of supportability. For example, a decision to use assembly language during development in the interest of some form of efficiency must also consider the cost of supporting the assembly language code over the entire readiness phase of the system's life cycle.

II. Software Readiness Planning

During the acquisition and development of avionics software, we are interested in the following readiness issues and must ensure that they are included in the Software Life Cycle Management Plan or its equivalent prior to contracting or tasking:

* Sufficiency and quality of the documentation
* The development methodology to be used
* The engineering standards to be used
* The programming language to be used
* The system test requirements
* Availability of system resources for making changes
* Sufficiency and adequacy of the support environment
* System transition planning

Documentation is clearly the most critical requirement in supporting large amounts of software. During a system's readiness phase, software changes will be required long after the original developer is no longer available. To make a successful change to an item of software, the engineer must know what that item was required to do, how it fits into the rest of the system, and how it actually works. He must then determine how the item must be changed, how to integrate the new code into the existing software, and what changes must be made to other items of software so the the new one will interface properly. Documentation is the only tool an engineer has available to decipher all of this information. Program Design Languages (PDL), flow charts, and program listings either do not contain the necessary information or it is too impractical to dig out the information. For example, software requirements and common data base definition usually are not found in those forms of documentation. High level design and interface definitions are extremely difficult to extract. PDLs, flow charts, and listings are best used in the area of understanding the as-built information.

A full range of documents are needed. Documents must be available during the readiness phase that contain accurate, complete definition of the software's requirements, design, interfaces, data bases, and the as-built description. In order to be able to continue to test the system, as well as test the subsequent changes to the system, documents containing the testing requirements and detailed test procedures must be available. The bit pattern of any firmware in the system must be documented and, naturally, operator and user instructions too. It is imperative that all of these required documents be produced concurrently with the software, since producing them by reverse engineering during the readiness phase is considerably more expensive than obtaining them under the original software development contract.

The methodology used to develop software has considerable impact on the supportability of software. A highly modular structure developed top down using the "information hiding" and "structured programming" methodologies is much more modifiable and changeable. The methods employed also impact other supportability requirements such as transportability and reuseability. These are important considerations that must be included in project planning activities.

When deciding on the engineering standards to be used during software development, supportability again must be considered. As discussed above, software is supported much more easily if it is developed using a high level language, such as Ada, rather than an assembly language. The quality assurance procedures must be oriented toward assuring satisfaction of readiness requirements. Often overlooked, but an extremely valuable supportability tool, is the history of the software. Items such as Unit Development Folders (UDF), software trouble reports, software change control documents, and test results that are a part of the software development process provide very useful historical information to a software engineer involved in the software change process.

Testing during software development impacts its supportability in several ways. Obviously, the quality of the test effort greatly influences the resulting reliability of the system. Beyond that, keep in mind that the software needs to be tested many times during its much longer readiness phase. After each change to the software, the change itself must be tested and a certain amount of system regression testing performed. Consequently, all of the test specifications, procedures, tools, environments, and reports (history) that were used in developing the software are also needed to support the system for the remainder of its life cycle. Planning must include provisions for providing these resources to the life cycle software support activity sufficiently in advance of system delivery.

Inherent in the concept of life cycle readiness of software is the making of changes to the software. These changes range from correcting minor errors to implementing major system enhancements. The implementation of a change usually consumes some amount of the memory, execution time, and/or I/O resources. Thus, these resources must be available in the target avionics system in order to effect the change. Adding these resources to a avionics system during its readiness phase may not be possible because of space, weight, power, cooling, or interference limitations. Planning before system development is necessary to insure the availability of these resources during the readiness phase of the life cycle. Many of the U.S. Department of Defense contracts for development of avionics systems are requiring a reserve in the availability of these computer system resources of 50% at the time of delivery to the Government.

Planning for the life cycle software support environment and the orderly transition of software support from the developer to the life cycle support activity are such important considerations that each will be discussed in subsequent sections.

III. Software Support Environment Planning

There are two software support environments to be considered. The one used during original development of the software, Developmental Software Support Environment (DSSE); and the one to be used for the remainder of the software's life cycle, Life Cycle Software Support Environment (LCSSE). Both must be planned in detail prior to the commencement of any project activities, especially contracting. In order to transfer successfully an avionics system from its developer to its life cycle support activity, these two environments must be compatible and the LCSSE must have the requisite capability. There are other important planning issues to be addressed before discussing the specifics of LCSSE planning.

Foremost is the true total cost of software development. This cost includes, not only the cost of developing the software, but also the cost of establishing the life cycle software support capability. A compiler provides a good example. If the software developer uses a compiler that is not a component of the LCSSE, one must be added. If the customer is forced to purchase the compiler, its cost must be added to the cost of software development and must be included in the project's budgeting activities. This situation is even more important in the case of a competitively awarded contract for the software development. The cost of establishing the life cycle software support capability should be taken into account when evaluating the proposals. What appears on the surface to be the most cost effective proposal may turn out to be the most expensive in the long run when the support software costs are included.

Related to the true cost of software development are the costs associated with data rights, when proprietary products are used. In a cost-plus type contract, who pays for the cost of using the proprietary product during software development? Is the proprietary product required in order to perform life cycle software support? If so, how much does the right to use the product in the LCSSE cost? Who supports the proprietary product if its owner goes out of business or decides to stop supporting the product? These are just some of the questions that must be resolved when an item is used during software development that has restrictions on its use. These answers are needed before the item is used and the customer has no choice but to pay whatever the cost.

To give visibility to the cost of software support items and their data rights and to assist in assuring supportability of the system, the development of a DSSE Plan is highly recommended. Ideally, the software developer prepares the DSSE Plan and then obtains the approval of the customer. Once both parties have agreed to the Plan, all subsequent changes and deviations to the Plan then require concurrence by both parties. The remaining paragraphs of this Section discuss the type of information that is typically contained in a DSSE Plan.

A. Introduction - A top level executive summary of the contents of the DSSE Plan. It should contain a brief overview of the DSSE, LCSSE, and the target computer system. This overview includes the relationships among these three systems. The differences between the DSSE and the LCSSE are to be highlighted. Requirements for the customer to provide any resources to the developer for inclusion in the DSSE are summarized herein. It is imperative that a general discussion of limited or restricted rights, licensing agreements, or any other legal limitations involved be presented at this point. This Section also provides a convenient opportunity to discuss any involvement by subcontractors and vendors.

B. Applicable Documents - The DSSE Plan is to include a reference to all documentation that is related to the DSSE or the supportability of the target system.

C. DSSE Description - A detailed description of the DSSE. The internal organization of the DSSE and its interfaces with all aspects of the software development process is to be included. The use of graphics to show work and data flows are very helpful in depicting the various relationships. Topics to be addressed as subsections are:

Host System Description - The host computer, its operating system, and peripheral devices.

Central Library - The organization of the central library, its file structure, data base management system, and interfaces.

Tool Set - An itemization and description of all tools available in the DSSE.

User/System Interface - Operating features of the DSSE and instructions for accessing and using the system.

DSSE Relationship to Target System - A description of all items in the DSSE as to how they relate to the target system under development.

D. DSSE Relationship to the LCSSE - A detailed discussion of the relationships between the two environments. Topics that must be addressed as subsections are:

DSSE/LCSSE differences - The detailed differences between the DSSE and the LCSSE. It is critical that the customer fully understand how these differences affect the compatibility of the two environments and what effect that has on the life cycle supportability of the target system under development.

Items Unique to the Target System - A listing of all items to be used in the DSSE that are unique to the target system to be developed.

Items Recommended to be Added to the LCSSE - This Subsection provides the developer an opportunity to propose the use of DSSE items that could be added to the LCSSE to enrich its capabilities.

Items to be Used Only in the DSSE - An itemization of those DSSE items that are not recommended for addition to the LCSSE.

E. DSSE Implementation - The developer's planned management procedures and implementation of the DSSE. Also to be included is information describing the DSSE Plan's relationship to all other project Plans such as Software Development, Quality Assurance, and Configuration Management. Topics that must be addressed as subsections are:

Items to be Furnished by the Customer - Those DSSE items that are required to be furnished by the customer.

Items that are Commercially Available - All DSSE items planned for use that are available commercially. If the use of any of these items involves proprietary considerations such as limited or restricted data rights or licenses, it is imperative that the customer be aware of that fact and consider the associated costs that he will incur in order to perform life cycle support of the target system.

Items that are Privately Developed - All DSSE items planned for use that are not available commercially but are available from private sources. If the use of any of these items involves proprietary considerations such as limited or restricted data rights or licenses, it is imperative that the customer be aware of that fact and consider the associated costs that he will incur in order to perform life cycle support of the target system.

Items that are to be Newly Developed - Those DSSE items that do not exist at the time the software development project is begun but must be developed concurrently in order to support the development effort.

F. DSSE Operation - A description of the procedures and controls to be used in managing the utilization of the environment. Also included must be a discussion of how all subsequent changes to the DSSE will be managed.

G. Target System Supportability - The ability to operate and support the target system is, in essence, the bottom line of the software development process. This Section addresses those issues that determine whether or not the target system can be supported. Topics that must be addressed as subsections are:

Addition of new Items to the LCSSE - During development of the target system, items of support software and perhaps hardware can be expected to be used that are not currently a part of the LCSSE. It may be necessary that some of these items be incorporated in the LCSSE in order to perform life cycle software support of the target system. There are two aspects to be considered. First, those items must be obtained for inclusion in the LCSSE. Secondly, those items must be operable in the LCSSE. Both subjects must be taken into account in the planning.

Execution Compatibility - It is very important that all software newly integrated into the LCSSE execute properly in its new environment.

Operation Compatibility - It is also important to ensure that all operations or functions that were available during development are also available the people who will be maintaining and supporting the system over the remainder of its life cycle.

Compatibility of Results - In order to support the software of new system it is necessary that the newly delivered software produce the same results when executed in the LCSSE that it did when it was developed and tested in the DSSE.

Compatibility Warranty - The software developer must be held responsible for ensuring that nothing is done during development that has an adverse effect on the new software's proper execution, thus its supportability, after delivery. It is not unreasonable to ask the developer to identify to the customer how he intends to demonstrate that the compatibility needed for life cycle supportability has been achieved. Addressing these compatibility issues in the DSSE Plan not only serves to insure that these important consideration are not overlooked, but it provides the vehicle for both the developer and his customer to jointly agree on how this compatibility will be demonstrated.

IV. Software Transition Planning

The next area that must be addressed in software readiness planning is the orderly transition of all software to be delivered from custody of the developer to custody of the customer. The transition includes all deliverables of the software development effort, the documentation and support tools as well as the computer programs. It is important that the system turn-over occurs on the prescribed date. To leave software support with the developer normally represents an unprogrammed cost growth or a delay in operational capability. A smooth, effective transition is one which results in minimal disruption of system support.

Critical to a smooth transition is that the customer be completely prepared to receive the new deliverables without a disruption in support to the new system. The customer must have adequate hardware and software facilities in place, and must have adequate personnel on board trained in the operation and maintenance of the new system. This could require additional LCSSE host computer system resources to be installed. The new system may require new support software to be added to the LCSSE such as simulators, to be available for testing of changes to the operational software. Various support personnel may need to attend specialized training courses. The LCSSE operating procedures may need to be revised. New facilities may need to be constructed or existing ones modified. To have available these types of capabilities at the time of scheduled system turn-over may necessitate long lead-time software developments, hardware acquisitions, and training, all of which require advance planning.

The transition plan should include the following considerations:

A. Responsibilities for accomplishment of significant transition activities need to be assigned to specific personnel.

B. Additional hardware required for inclusion in the LCSSE must be itemized. Similarly, the additional support software required for inclusion in the LCSSE must also be itemized.

C. Additional manpower, numbers and skills, required for support of the new system along with all related training requirements need to be identified.

D. The logistics involved in the physical transfer of the new system from the contractor's facility to the customer's facility needs to be programmed.

E. Contracting for construction or facility modification requires programming (funding too).

This mandatory planning may be contained in a separate transition plan or could be included in the DSSE Plan. As noted in an earlier paragraph, it can be seen that the central focus of the transition plan should be the schedule of milestones and events leading up to the turn-over. The transition's critical path should be identified, paying particular attention to the lead-times associated with the accomplishment of each item.

V. Contracting For Software Readiness

Over the past decade there has been a dramatic increase in the acquisition and deployment of digital avionics systems. Although some commonality has been achieved, the diversity of requirements has resulted in a large number of unique computer systems. Each unique computer system brings with it its own set of life cycle software readiness requirements. The complexity of these digital avionics systems requires the life cycle software support activity to provide a full range of engineering services, including the integration of the operational software with the other elements of the avionics system.

When several avionics systems are assigned to a single life cycle software support activity to gain the economies of centralization, the software support environment must support all common functional requirements and capabilities, as well as the unique support requirements of each system. This workload demands an extremely large software support environment base to maintain and enhance all these systems. Compounding the situation is the fact that these systems are being developed by many different contractors.

A standard subset of each life cycle software support environment needs to be identified and software development contractors should be tasked to be compatible with this portion of the life cycle software support environment that exists at the designated life cycle software support activity. As each environment matures, the standard environment is expected to grow, and thus increases the requirement for standardized and transportable support software. The life cycle software support activities must identify their support requirements and uniformly task the software development contractors to be compatible with their environment before the new software is delivered.

Acquisition managers, when contracting for systems that include the development of software, are often unfamiliar with the requirements for life cycle software readiness. They are usually unaware of the various components of the software support environment needed in order to perform the life cycle software support functions during the readiness phase of the system's life.

For the acquisition manager, use of a contracting standard is the most effective way to solve many of the software readiness problems. Such a standard provides a common requirement for all contracts that contain software development and provides a vehicle to ensure compatibility with a designated life cycle software support environment. It is also a tool for program managers to use in writing their contracts to make sure that this important area receives proper contractual attention. It also allows the life cycle software support activities to identify, prior to selection of the software development contractor, their particular requirements and to task the software development contractor to be compatible with the existing standardized life cycle software support environment.

In answer to the need for such a contacting standard, the U.S. Army has developed DOD-STD-1467 (AR), "Software Support Environment", and an associated set of support software documentation requirements to address fully the requirements for life cycle software readiness. This Standard is intended to be a contractual companion to DOD-STD-1679A, "Software Development" and DOD-STD-2167, "Defense System Software Development". DOD-STD-1467 is to developing and acquiring support software what these other two Standards are to developing and acquiring operational software. DOD-STD-1467 ensures that all issues of software readiness are properly addressed in contracted software development efforts, and it provides the mechanisms to ensure the existence of a complete life cycle software support capability for contractually deliverable software upon its entry into the operational inventory. Although DOD-STD-1467 was developed for the Army, it provides the basis and structure for any organization to ensure the life cycle supportability of their software.

The Standard is designed to recognize the needs and constraints of existing life cycle software support activities and, at the same time, to allow the contractor flexibility to develop software and manage the contract in accordance with his best judgement and practices. Accordingly, DOD-STD-1467 does not dictate the approach to be used by the contractor.

The contracting activity normally identifies, in the request for proposal, the designated life cycle software support activity and any of its items that are designated for use by the contractor. Subject to the constraints imposed by the contracting activity, the contractor may propose to use the existing resources of the life cycle software support activity, to use the contractor's own resources (either existing or to be developed), or to select from a wide range of options in between. The contractor then identifies his selected approach in the proposal for the contracted software effort. The contractor's approach is considered during source selection.

There is a 5-step approach to using DOD-STD-1467 in a contract: 1) establishment of a jointly approved DSSE Plan; 2) implementation of that Plan during software development; 3) smooth transition of all contractually deliverable software to the LCSSE; 4) demonstrated supportability of all contractually deliverable software in the LCSSE itself; and 5) ensuring that the new items of software operate correctly in the LCSSE.

Inadequate provisions for protecting the U.S. Army's rights to technical data and computer software necessary for operation and support of its weapon systems historically have been responsible for decreased operational and support capabilities. In some cases, considerable additional expense was necessary to procure or develop the information required by the Army to sustain the operation and support of its operational systems. As a general rule, the contract guarantees the Army unlimited rights to all technical data and software specified in the contract's Delivery Schedule. However, the acceptance of software with restricted rights may be necessary for the Army to obtain rights to use contractor-owned or state-of-the-art software that improve the operation and efficiency of the life cycle software support activities. In addition, to maintain the currency of the software support organizations, it may be in the best interests of the contracting activity to accept limits and restrictions on support software. This acceptance should encourage the use of new or improved software support items by contractors and ensure the accessibility of these items to the life cycle software support activities.

DOD-STD-1467 defines the responsibilities of the contractor to obtain approval to deliver software with restrictions. This prior approval encompasses all software or documentation required to operate and support the contractually deliverable software over the readiness phase of its life cycle. The specifics of types of software and their limitations are addressed in the contractor's proposal, negotiated and resolved during proposal evaluation and contract negotiation, and included in the resulting contract.

An Ada Programming Support Environment (APSE) is a good example of the type of life cycle support environment envisioned by DOD-STD-1467. Contracting activities may designate or furnish the APSE to be included in the contractor's DSSE. The Standard supports the APSE concept of providing the identification and adoption of new tools for a selective (LCSSE) growth; and allows the contractor freedom to incorporate new (APSE) tools in his DSSE. Since the DSSE is required to be compatible with the LCSSE APSE, the augmented supportability of the system by APSE tools can be operationally verified and warranted. This allows for continued growth and improvement of APSE capabilities.

In summary, the use of DOD-STD-1467 forces as many decisions as possible into the proposal and subsequent contract; requires prior customer approval of support software used to develop the operational software; requires the contractor to ensure life cycle supportability of operational software; and minimizes any additional cost growth to the actual software development contract. Also, the use of DOD-STD-1467 minimizes any possible conflicts regarding the "rights in data" problem and it provides a "measuring tool" for determining the life cycle readiness of the operational software.

VI. Summary

Planning for software supportability during the readiness phase of a digital avionics system's life cycle has often been neglected. Consuming over 75% of the total cost of the system, it is very important that life cycle software supportability be as efficient and cost-effective as possible. The supportability of software is completely determined during it development. Consequently, planning for supportability must begin prior to contracting for software development. The DSSE Plan provides a good, convenient tool for this purpose and DOD-STD-1467 provides the contracting tool. This planning must provide for the efficient turn-over of the software from developer to customer. Life cycle software supportability demands more up-front attention than it historically has received.

AN AVIONICS SOFTWARE EXPERT SYSTEM DESIGN

Judy L. Brink
Control Data Corporation
8800 Queen Avenue South
P.O. Box 1305, BLCS1R
Minneapolis, MN 55440-1305
USA

Ronald B. Haynes
Rome Air Development Center / IRR
Griffiss AFB, NY 13441-5700
USA

SUMMARY

The scope of this paper is to define an expert software system, implemented in Ada, for avionics in the 1990's, using a table-driven design that was developed in a demonstration program as the prototype, in order to support tactical decision making applications. As a synopsis of what follows we will: point out the relevancy of such a paper to the avionics community; give an historical example of a table-driven design in the area of Reconnaissance Management Systems; describe the design logic behind this historical example by providing a single thread through the system; discuss the conceptual enhancements that need to be folded on top of this historical example, as an additional layer, in order to produce an Expert Software System; and provide a summary highlighting some reasons the avionics community should move in this direction from a software perspective.

1.0 INTRODUCTION

The orientation of this paper is from the user perspective and will develop an approach towards a user friendly Man-Machine Interface in the avionics arena. This paper is relevant to the AVP Symposium because it addresses the avionics life-cycle process using the Advanced Tactical Air Reconnaissance System (ATARS) as a practical example. The primary focus of this paper falls under the Application section, specifically, Tactical Decision Making in the area of Expert Systems, however, the paper also touches other relevant topics such as Expert Machines, Man-Machine Interfaces, and Ada as an application language. Currently, there is a need for avionics systems that can make tactical decisions with minimal operator intervention. In today's high-threat environments, modern tactical reconnaissance requires adaptive, flexible systems for mission success. Crew members need to focus on navigation, aircraft control, and threat avoidance, and a semi-intelligent system will facilitate that concentration.

The concept for such an Expert System evolved from the video reconnaissance system prototype developed by Control Data Corporation. This prototype was flown by the United States Air Force (USAF) during both the 1986 Near-Real-Time Reconnaissance F-16 Flight Test Demonstration and Concept Validation (F-16 Recce) program at Edwards Air Force Base, California, and the 1987 RF-4C Advanced Tactical Sensor Demonstration (ATSD) program at Eglin Air Force Base, Florida. The demonstrations were designed to test the operational as well as tactical deployment capabilities of Control Data's real-time Reconnaissance Management System (RMS), a sensor data collection, review, and transmission system. This system supported both manual and semi-automatic target acquisition and data link capabilities. The overall system evaluation was performed by Rome Air Development Center (RADC). Before discussing the RMS prototype, a brief discussion of reconnaissance is in order.

2.0 BACKGROUND

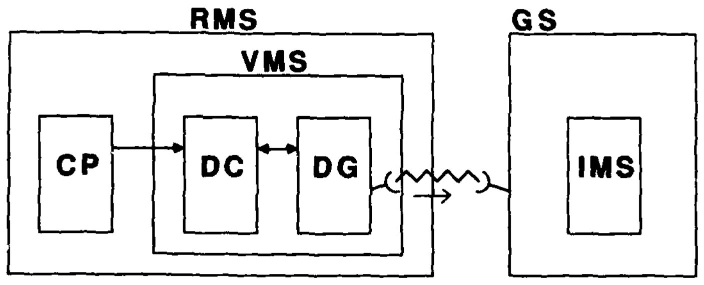
Historically, military commanders have been dependent upon airborne reconnaissance photographs for intelligence decision making in regard to strike force operations. However, film processing is not only cumbersome but also very time consuming. Developing film in the battlefield involves transporting bulky equipment, processing chemicals, and excessive amounts of fresh water. Of greater concern, however, is the time involved with film processing. An area classified as a high-value target may no longer be occupied by the time the film is processed and analyzed. With the high mobility of today's enemy forces, commanders need real-time information in order to make accurate tactical decisions regarding deployment of scarce resources. Consequently, a more timely method is needed in order to provide that capability.

The U.S. Air Force sought to replace the film-based systems with an integrated reconnaissance system that would include a video management system, electro-optical sensors, a data recorder, a data link, and a digital imagery ground exploitation system. In response, Control Data developed the airborne RMS and the ground-based Imagery Management System (IMS). The RMS is an airborne video management system that interfaces to multiple sensors, either electro-optical or infrared, a data recorder, and a data link system. The RMS provides the in-flight capability to record, recall, review, and transmit pertinent sensor imagery in near real-time.

For both the F-16 Recce and RF-4C ATSD demonstrations, the in-flight data link operations and post-flight data analysis were supported by RADC's Ground Station system. The Ground Station is a mobile ground processing and exploitation system that provides photo and imagery interpreters with the capability to perform in-depth analysis of post-mission reconnaissance data. Control Data designed and developed the Imagery Management System (IMS) based on the airborne RMS in order to provide the front-end to the Ground Station that is necessary for real-time processing of image data from the data link and airborne tapes into a video display format. For both demonstrations, the IMS provided the imagery interpreters with a more extensive video tape display and control capability than was provided by the airborne RMS. RADC evaluated the IMS, and results proved not only that the IMS worked extremely well, but also that softcopy digital reconnaissance is a valid concept.

The previously mentioned demonstrations evaluated by RADC were precursory studies for the Advanced Tactical Air Reconnaissance System (ATARS) program. The ATARS program involves: upgrading the service's current RF-4C reconnaissance fleets with electro-optical based video reconnaissance systems; developing a follow-on podded version for the reconnaissance aircraft to replace the RF-4C; and future equipping of an unmanned air reconnaissance vehicle. The demonstrations successfully validated the feasibility of developing an electro-optical/infrared sensor based video system with excellent results.

Control Data was the prime system integrator for the RF-4C ATSD program and was responsible for the hardware and software development for the RMS and the IMS. Figure 1 shows the RF-4C overall major interfaces at a functional block diagram level. For this demonstration, the RMS consisted of a Data Collector (DC), a Display Generator (DG), and a Control Panel (CP). The Data Collector accepted sensor inputs and combined them with aircraft navigational inputs into a format used for real-time cockpit display and storage on a data recorder. The Display Generator accepted live or recorded inputs and processed the input data for presentation on a video display or for data link to the Ground Station. The Control Panel provided the interface between the aircraft navigational computer and the Data Collector as well as providing the interface between the operator and the RMS.



Legend

RMS	=	Reconnaissance Management System
VMS	=	Video Management System
CP	=	Control Panel
DC	=	Data Collector
DG	=	Display Generator
IMS	=	Imagery Management System
GS	=	Ground Station

Figure 1. RF-4C Functional Block Diagram

For the RF-4C flight tests, the RMS had four major modes of operation: System Status, Sensor Data Acquisition, Sensor Data Review, and Data Link. The System Status mode showed the viability of the airborne RMS system at any point during the mission, either on the ground or in the air. The Sensor Data Acquisition mode provided the capability to record sensor imagery data and mark targets of interest for later review. The Sensor Data Review mode permitted in-flight review of the recorded sensor imagery and manual selection of sensor imagery to be data linked to the ground. Finally, the Data Link mode provided wide or narrow band transmission of sensor imagery data to a ground station.

The Data Acquisition mode performed either manually or semi-automatically. In the manual acquisition mode, the operator would select the primary and secondary sensors most appropriate for coverage of the desired target, turn recorders on and off over the target area, and mark the target as it was overflown. In the semi-automatic mode, the operator would still select the sensors to use, but recording and marking functions were performed by the RMS matching aircraft geolocation coordinates with target coordinates preset by the operator. Since acquisition of target data generally occurs in hostile territory, the automatic Acquire mode is important because it eliminates the distraction of system operations when full attention is best focused on tactical maneuvers. The automatic Acquisition mode functions also demonstrated the feasibility of employing an RMS on a single seat aircraft or on an unmanned vehicle platform.

The Acquisition mode also provided a variety of useful video display options such as Pause, which froze the current image on the display; Invert, which inverted black video to white and vice versa; Compress, which compressed the video in the along-track direction, effectively slowing image motion; and Target Data, which superimposed current aircraft information over the video imagery.

The Sensor Data Review mode enabled the operator to recall and replay portions of recorded imagery that were marked during acquisition at either a real-time rate or a slower, fixed rate. Magnification of imagery could also be achieved with full resolution since the system recorded high resolution sensor data. In this mode, the operator would also select the most important segments of imagery, magnified or unmagnified, for data linking to the ground. Since review of sensor data is operator intensive and time consuming, the Review mode is an optional feature and is generally most useful in friendly territory when time permits.

3.0 TABLE-DRIVEN DESIGN EXAMPLE

The RF-4C Control Panel (CP in Figure 1), which was the Man-Machine Interface for the RF-4C RMS, was designed as a state/table-driven software package. Essentially, the software architecture mirrored the Control Panel, which was comprised of programmable display pushbuttons (PDPs), discrete buttons, and a keypad as shown in Figure 2.

The various mission modes (e.g., data acquisition), as well as options within a given mission mode (e.g., sensor selection), were encapsulated in the form of tables within the processor memory. For example, Figure 3 depicts the operator's options as a series of tables. Table 3 is the currently "executing" table in memory, which, in turn, reflects the operator's current menu of choices. This illustration is for the Acquire Mode as shown in Figure 2.

For each selection that the operator makes with respect to the menu presented in Figure 2, the following sequence occurs: first, the software is vectored to the appropriate offset within the current table (Figure 3) by the pushed button interrupt type (e.g., P6 which corresponds to the COMPRESS option in Figure 2's menu); second, the validity of that selection is checked by the corresponding boolean entry for this selection (e.g., P6 is a valid selection); next, the action specified for this selection, which in this case is to compress the real-time imagery on the cockpit display, is then executed within a High Order Language (HOL) CASE construct as shown in Figure 4 below; then, new messages are displayed on the Control Panel per the associated message index; and finally, the next table index field is used to update the current table in memory to correspond to the next menu. This process is then repeated until the mission is over.

In addition, the system also supported the capability to automatically acquire and data link target information. At the beginning of the mission the crew member would enter the target information, in terms of geolocation coordinates, into the data base. If the operator then selected the automatic target acquisition option, the RMS would process the incoming navigational data against the data base target data parameters. In essence, the system drew two imaginary concentric circles around the target, the outer circle having a radius of eight miles and the inner circle having a radius of about three miles. When the outer circle was pierced by the aircraft, the system would turn on the recorder just as if the operator had depressed the record button (Figure 2). If the aircraft proceeded to enter the inner circle around the target, the system would then mark the target as if the operator had also depressed the target mark button. Marked video and associated aircraft data were saved in the data base, and subsequently data linked to the ground station, again, as if the operator had entered the Review mode and pushed the data link mark button to select imagery.

This capability proved extremely useful because it freed the operator from the manual intervention mode. Reducing the number of selections an operator has to make correspondingly reduces the potential for errors that can occur, which is more likely during tactical maneuvers. Of course, this capability had some limitations. For example, since the targets were determined a priori, there was no automatic method for acquiring a target of opportunity. That is, if the crew member determined that a target not programmed in the data base was nevertheless a target of interest, then the operator had to enter the target data parameters manually after manually marking it. The chance for errors in this manual mode increase with such variables as speed, target mobility, etc. The automatic acquisition function demonstrated utility, but the concept requires some additional refinement to be truly automatic. This increased capability is discussed further in this paper during the treatment of an Expert System.

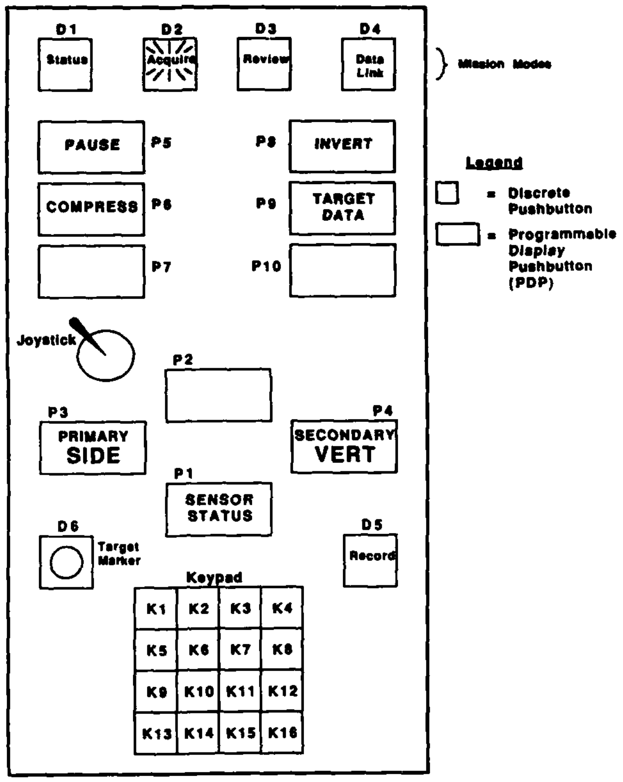


Figure 2. Control Panel Operator Inputs

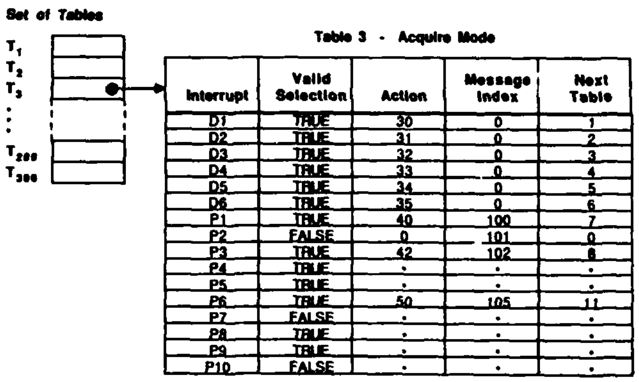

Set of Tables
T_1
T_2
T_3
T_{299}
T_{300}

Table 3 - Acquire Mode

Interrupt	Valid Selection	Action	Message Index	Next Table
D1	TRUE	30	0	1
D2	TRUE	31	0	2
D3	TRUE	32	0	3
D4	TRUE	33	0	4
D5	TRUE	34	0	5
D6	TRUE	35	0	6
P1	TRUE	40	100	7
P2	FALSE	0	101	0
P3	TRUE	42	102	8
P4	TRUE	•	•	•
P5	TRUE	•	•	•
P6	TRUE	50	105	11
P7	FALSE	•	•	•
P8	TRUE	•	•	•
P9	TRUE	•	•	•
P10	FALSE	•	•	•

Figure 3. Software Representation of Figure 2's Menu

```
case ACTION of
   0 :  begin
           { do nothing }
        end;

   1 :  begin
           { read temperature probes }
        end;
        { .........
         .........
         ......... }
  50 :  begin
           { compress imagery }
        end;

        { ......... }
end; { case }
```

Figure 4. HOL Construct for Action Field in Figure 3

The mission mode tables were statically constructed and could not be altered. Prior to the mission flights the tables were built in assembly language and burned into Erasable Programmable Read-Only Memory (EPROM). The tables reflected the different scenarios that were specified by the system requirements. If a requirement for a particular sequence changed, then the tables were modified. The code was only changed if a new action was specified, but the changes were localized to the CASE construct (Figure 4). Consequently, validating an operator selection, performing an action, updating the panel PDPs, and transitioning to the next table were predetermined.

This behavioristic, modular approach facilitated system requirement changes, integration, and software maintenance. The software was not plagued by the traditional problems associated with non-table driven designs. Requirement modifications did not ripple all the way through the software system, which saved time during the early development phase. The integration problems that lengthen the later development phase of software programs built on traditional logic (e.g., IF ... THEN ... ELSE ...) were avoided. For example, anyone experienced with real-time embedded software systems development must be familiar with the race conditions inherent in "flag" driven software. In addition, there was minimal maintenance requirements in terms of contractor support personnel, which is cost effective when one considers the burgeoning costs associated with software maintenance. Finally, this approach simplified the training and transition cycle from contractor to customer in that new personnel did not have to learn the system by walking through all the attendant code and documentation.

The Control Panel software, which was written in Pascal, "executed" tables. In effect, the numbers contained in the tables represented pseudo operation codes, which extended the processor's instruction set. This approach is less dependent on hardware architectures and language processors. The application happened to use an Intel 80188 processor, but could just as easily have used a Motorola 6809 with the code generated in the C programming language. The design would not be impacted if the prototype went into production on the standard 1750 processor using Ada, the language mandated by the Department of Defense. The portability of this approach across processors and languages is clearly a desirable trait.

However, despite the positive characteristics of table-driven software designs, there are still some shortcomings associated with such an approach. For example, as previously mentioned under the discussion of the automatic target marking function, the system was not truly automated since it could not cleanly handle the "target of opportunity" case. In the manual mode of operation every scenario was predetermined. Changing a course of action, based on a new set of variables not previously encountered, was not possible. The system could not infer the next set of operations in this case. Finally, in the manual mode of operation, a great amount of interaction was required on the part of the operator. A semi-intelligent system is needed to: produce an even more user friendly Man-Machine Interface; provide a fully automated acquisition mode, not just automatic target marking; expand previous automatic target marking functions to handle "targets of opportunity"; and deduce a new sequence of operations when encountered with a new set of variables.

4.0 EXPERT SYSTEM CONCEPT

Our concept of such an Expert System would be built on the table-driven design developed for the RF-4C demonstration program. Subsequent avionics systems (e.g., ATARS) could be based on this approach with the following progressions: an interpreter would be added that could build intelligent tables, i.e., tables that contain decision making properties; the application would be generated in Ada; and the tables could be constructed dynamically as a set of executable packages by manual and/or voice intervention. For instance, the operator could select a mission scenario configuration, which would cause the construction of the required set of tactics in tabular format to be interpreted. Since we have not designed such a system yet, a detailed discussion is not possible. However, this paper will attempt to sketch our approach. The layered software depicted in Figure 5 represents a high-level view of our approach. Please note that the outer layer, the interpreter, was not present in the RF-4C demonstration program.

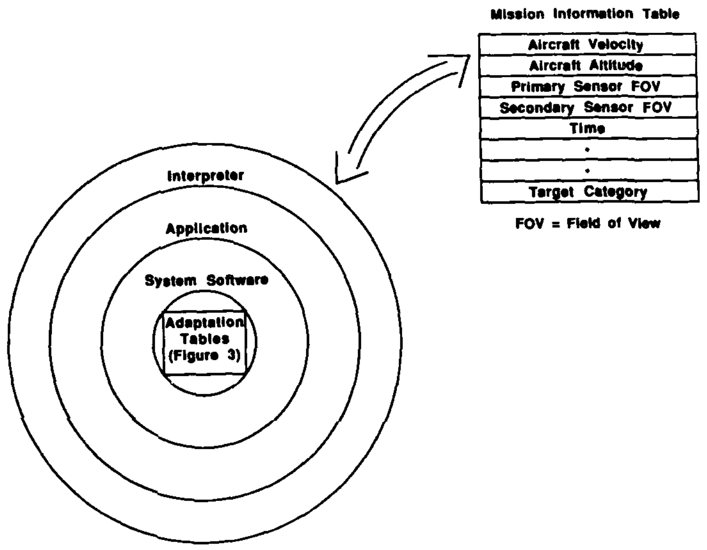


Figure 5. Layered Software Expert Systems Approach

Tables would be at the heart of this system just as in the previous system, but with some notable differences. First, we envision two different kinds of tables. The first set of tables, namely, Adaptation Tables, would be identical to the RF-4C approach. That is, the system would support the RF-4C static mission scenarios as a subset of it's overall design. There are many reasons for maintaining the architectural design from our previous example. First, there is no foreseeable replacement for real intelligence, which is the human interface. Therefore, the system must be flexible to provide the operator a manual mode. Second, if the automated mode breaks down, then a fully manual default mode must be available. Finally, the Review Mode does not lend itself to an Expert System implementation. The system cannot: "know" that the operator wants to review any imagery, since this mode is optional; review a particular piece of imagery; or select areas to magnify; for example. Therefore, we have targeted the Acquisition and Data Link modes as candidates for knowledge-based software systems.

The second set of tables, namely, Mission Information Tables, would be a knowledge base containing current information required by the Interpreter (Figure 5) to automate the Acquisition and Data Link mission modes. The data entries presented in the Mission Information Tables are not complete, but highlight the sort of data needed in order to modify the pregenerated Adaptation Tables in the Acquire Mode. Parametric data, such as aircraft data provided in real-time, would allow the Interpreter to make decisions about best sensor selection. Supplied data, such as a sensor's Field of View (FOV), would augment the decision making process for the Interpreter.

The Interpreter would execute as a periodic (e.g., 1 second) real-time monitor. It would repeatedly examine the Mission Information Tables to interpret the current situation and construct the appropriate Acquire Mission sequence. For example, if the operator was flying low and fast, the Interpreter would build a menu like Figure 2 and "depress" the COMPRESS PDP in order to slow the imagery for viewing purposes. As another example, if the target coordinates were perpendicular to the line of flight, the Interpreter would select a side oblique camera as the primary sensor, since it would provide the best coverage. In essence, the Interpreter is constantly selecting the Acquire Mode and making determinations about which sensor will provide the best primary/secondary coverage, or whether imagery should be compressed for real-time viewing, for example.

The automatic target marking function would be handled analogously to the relationship between the RF-4C prototype and the semi-intelligent system proposed for this ATARS application. That is, primary targets would still be identified in the data base prior to flight in terms of geolocation. The Interpreter would compare these preset coordinates against current aircraft navigational data in order to determine when to mark an area of interest for subsequent review and data link. This would be the baseline automatic target acquisition function for an ATARS system.

In addition, areas of interest could be denoted by the operator in terms of time. Since the Interpreter would execute as a periodic task it could effectively mark areas along with the specified time coordinates. The only preset parameter needed would be the frequency interval, which would specify how often to mark areas of interest. For example, if the periodic interval was specified as 5 seconds, the system would store time, location on tape, and current video image pointers in the data base. These set of marks, based on time, would be construed by the system as a secondary set of marks. The operator could then recall areas of interest via time, instead of geolocation coordinates, if so desired. By definition, this approach solves the "target of opportunity" case in the sense that there are no specific targets or "targets of opportunity".

In addition to using geolocation, using time as a coordinate permits the construction of a video road map. In addition to the current exhaustive post-mission data analysis provided by RADC's mobile ground station, it could generate a quick-look report by reviewing imagery based mainly on time interval snapshots taken during the airborne mission. This strip chart could be produced very quickly to provide information to imagery interpreters to make tactical decisions about targets in terms of rank, category, etc.

Programming the application in Ada adds another dimension to the RF-4C prototype baseline for this ATARS application program. The RF-4C program was implemented in Pascal. For purposes of this discussion Ada can be construed roughly as a superset of Pascal. However, Ada adds some essential features not provided by Pascal. The use of Ada as the HOL within this system will influence the design processes. The primary benefits of the language will be it's package structure and strong data typing, which provide a highly modular design. The semi-intelligent system would be designed in a layered structure to ensure a flexible approach to software development, while fully utilizing the benefits of Ada as the implementation language.

The tables referenced in this paper can be constructed as a set of executable packages. In turn, the development of the various layers of the ATARS program (Figure 5) can proceed in a top-down, modular fashion with strong interface type checking between the layers and the tables. Our experience has shown that there is a marked savings in effort during the later phases of a program (testing, integration, and maintenance) by using Ada as opposed to other languages. The reason for this savings is two-fold: first, many of the errors that occur during the integration and test phase using traditional languages (e.g., FORTRAN) are at the interface level between software modules; second, the top-down approach using traditional languages does not permit early testing. Ada's strong syntactical type checking between modules prevents the first problem. Ada's separate module compilation (body and specification) permits bottom-up testing during the early development phase, which provides a parallel approach in terms of designing to test early.

5.0 CONCLUSION

In summary, we have taken the RMS Control Panel table-driven design from the RF-4C demonstration and discussed conceptual enhancements necessary to produce an expert system. First, we added an Interpreter to dynamically build intelligent tables. These tables, namely, Adaptation Tables would be an extension of the static RF-4C tables. We also introduced Mission Information Tables that would be a knowledge base to record current parameters used by the Interpreter to fully automate modes of operation. The expert system would control the system configuration and manage the mission modes by performing actions consistent with the current set of tactics.

The concept for this expert system is based on the premise that modern tactical reconnaissance requires adaptive, flexible systems for mission success. We used the RMS Control Panel as a practical expert system application since it has relevance in the avionics community right now. Currently, in the ATARS program, there is a need for an RMS to interface to multiple vehicle platforms, manned and unmanned. Since the expert system provides fully manual and automated modes of operation, this design concept will facilitate that need, and will reduce the cost of software development from the design level through maintenance in the avionics software life-cycle process.

DISCUSSION

R.W.MacPherson, Ca

How do the pilots feel about never having the same menu appear twice?

Author's Reply

The panel always defaulted to the same menu. The demonstration panel was designed jointly with the USAF backseat operators that flew the test missions and those operators liked the panel. The functions on the menu appear in the same position on the panel during the mission but may reflect a different state of the function.

W.Mansel, Ge

Could the pilot in the automatic acquisition mode decide to initiate an attack when having a target acquired or was it a full automatic attack mode?

Author's Reply

The system performed only acquisition (and data link) of reconnaissance data. It had no attack mode. For target acquisition, there was both a manual and automatic mode. The operator would select the automatic mode prior to entering the target area.

BUILT-IN SELF TEST

B. Jansen
Océ Nederland BV
Research and Development (EE1)
Postbus 101
5900 MA Venlo
The Netherlands

A.J. van de Goor
Delft University of Technology
Faculty of Electrical Engineering
Section: Computer Architecture
Mekelweg 4
2628 CD Delft
The Netherlands

ABSTRACT

Because of the increasing complexity of digital circuits, it is becoming more and more difficult to determine whether a circuit is correct or faulty. Faults in a circuit can hardly be detected just by looking at the outside what the reaction of the circuit is to a certain input sequence. Fault tolerant computing can be a solution. Built-In Self Test (BIST) techniques can also be used to verify whether the circuit is correct, not only during normal operation (e.g. every time upon power-on), but also during the early development periods. The result of using BIST techniques is a considerable reduction of the time between design and the final product, and a reduction of maintenance time and cost.

BIST is a test method of which the circuit (on a board, a chip or a part of a chip) can separate itself from the surrounding logic, and perform a (self) test. After the self test, the circuit reports to the surrounding logic whether it is correct (a so-called go/nogo test). The advantage of BIST is that it is a universal and systematic test method with a solid mathematical foundation. Based on the stuck-at fault model, it is possible to compute the fault coverage, which is the number of faults detected by the BIST method.

In this paper the theory of BIST is described. A circuit is divided into combinational and sequential parts, which are tested separately. The sequential parts are tested with a so-called scan-path test. Alternative test methods to test the combinational parts are described, and one of these methods (using pseudo-random pattern generators and CRC signature analysers) is described in more detail. The method to compute the number of patterns needed to detect all faults with a certain probability as function of complexity of the circuit, is given. The theory of CRC signature analysers, and the probability of masking are also described and illustrated with some (small) examples, which can directly be used in practice.

1 INTRODUCTION

Over the years many different approaches to testing digital circuits have been developed. One of these approaches is Built-In Self Test (BIST). BIST is becoming more and more important, because of the increasing complexity of VLSI-circuits. These circuits can hardly be tested just by looking at the outside what its reaction is to a certain input sequence. There has to be a circuit inside the chip, which reports whether the chip is correct.

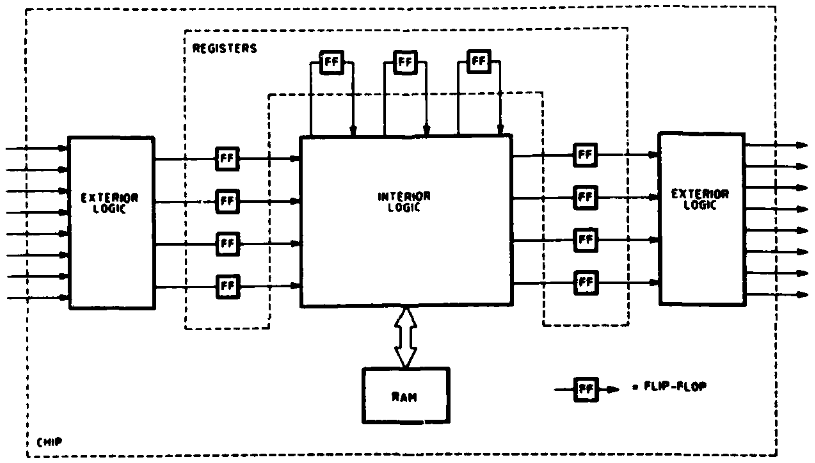


Figure 1: The four parts in which a circuit can be divided

A circuit (almost every circuit) can be divided into four parts (see Figure 1):

1. **registers**
   Registers comprise all memory elements, e.g. registers, latches, flip-flops, but not memory-arrays like RAM.
2. **interior logic**
   Interior logic is that part of the combinational logic which is surrounded by registers.
3. **exterior logic**
   The combinational logic which is not surrounded by registers is called the exterior logic. An example of exterior logic are the I/O buffers of a chip.
4. **memory**
   The last part which can be distinguished consists of memory arrays (e.g. RAM/ROM).

Testing the above four parts requires different test methods. These methods are:

1. **SCAN-PATH TESTING**
   This method is used to test the registers of the circuit. It will be described in Section 2.
2. **INTERIOR LOGIC TEST**
   The interior logic is tested with a interior logic self test method, see Section 3.
3. **BOUNDARY SCAN TEST**
   The exterior logic is tested with a test technique called Boundary Scan testing, see Section 4.
4. **MEMORY TEST**
   Memory (RAM/ROM) can be tested with the same test-method as registers, but this would result in too much hardware-overhead, or too much test-time. It therefore is tested with specific RAM/ROM tests, which are not described in this paper. For more information see [1], [2], [3] and [4].

The original circuit is configured such that during self test, a test-controller controls the data and the control-signals of the circuit. The test-controller is a so-called "state-machine" (a sequencer), which is brought into several states during the self test, depending on the previous state and the values of the incoming signals. This "state-machine" controls each stage of the self test and reports the test-result to the "outside world", i.e. the logic that started the self test of the circuit.

Note that the partitioning described above (i.e. the division of the circuit into registers, interior logic, exterior logic and memory), is not only useful to test a chip, but also to perform a self test for a board, or a part of a chip.

2 SCAN-PATH TESTING

The first thing one has to know before the other parts of the circuit are tested, is whether the registers are functioning. This is important, because these registers are used in the tests for the remaining parts of the circuit. The registers are tested using a test technique called *scan-path testing*.

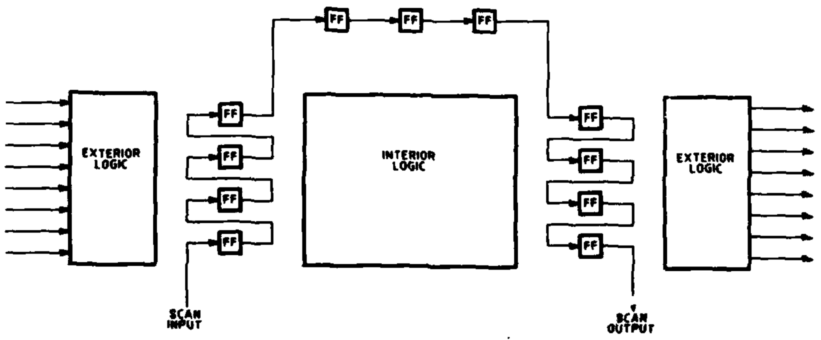


Figure 2: The registers of a circuit connected as a string

During a scan-path test the registers are connected with each other to form a string, called scan-path (see Figure 2), through which a certain pattern is shifted. When the same pattern appears at the output, all registers of the string work correctly. A register should be re-configured such that during the scan-path test phase the pattern can be shifted through the registers. These re-configured registers are called scan-path elements. A scan-path element is a flip-flop or a latch with some extra logic, needed to perform a normal function during normal operation and to perform a scan-path function during the scan-path test. An implementation is given in Figure 3.

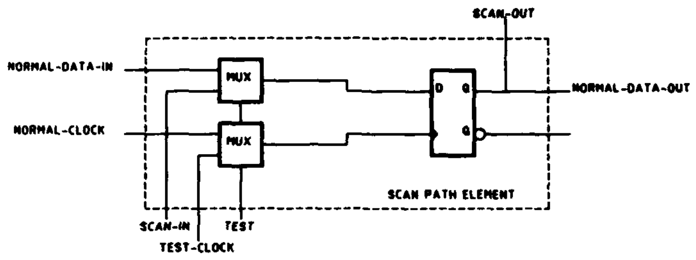


Figure 3: A scan-path element

During normal operation, the register is controlled by the "normal-clock"-signal, and the "normal-data-in" input is selected. During the scan-path test phase the signal "test" is active, the register is controlled by the "test-clock"-signal and the data from the previous scan-path element is used as data-input (through the "scan-in"-signal). The scan-path elements are connected with each other during the test phase by connecting "scan-out" of the previous with "scan-in" of the next scan-path element. The test-controller controls the signals "test-clock" and "test". It also sends a certain pattern into the first, and checks the patterns coming from the last scan-path element of the string. When they are equal the registers are correct.

Now that one knows how the scan-path elements look like, one has to know which patterns should be shifted through the register-string in order to be sure of a correct working of the memory elements. Before this question can be answered, one has to find out what faults can occur in a register. An enumeration of these faults is given below:

1. a register is stuck-at-one
2. a register is stuck-at-sero
3. a register cannot undergo a 0 → 1 transition
4. a register cannot undergo a 0 → 0 transition
5. a register cannot undergo a 1 → 0 transition
6. a register cannot undergo a 1 → 1 transition

In order to test these faults, the following pattern can be shifted through the scan-path elements:

0011001100110011001100 11 · · ·

The four-bit pattern 0011 is repeated until it has reached the last scan-path element. First a '0' is shifted into the string in order to test whether a register is stuck-at-one. The second '0' which is shifted in the string is to check whether the register can undergo a 0 → 0 transition. Then a '1' is shifted in the string. This is done to check whether the register is stuck-at-sero, and to check whether the register can undergo a 0 → 1 transition. The second '1' which is shifted in checks whether the register can undergo a 1 → 1 transition. Then the pattern is repeated, so the next bit which is shifted in is a '0', so also the 1 → 0 transition is tested (see also [5]).

3 INTERIOR LOGIC TEST

A general model for the interior logic test is shown in Figure 4.

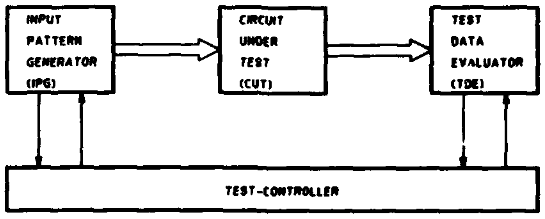


Figure 4: Global model for the interior logic test

This global model of the interior logic test is generally accepted. The Input Pattern Generator (IPG) generates several patterns, which result in a certain response on the output lines of the Circuit Under Test (CUT). The Test Data Evaluator (TDE) evaluates the response and determines whether the CUT is correct. The test-controller controls the IPG and the

TDE. The CUT is the interior logic, containing no registers, so the CUT is a combinational circuit: a pattern on the input of the CUT affects the output-lines directly.

In this section the following subjects will be discussed:

1. **FAULT MODELS**
   Before the Input Pattern Generators can be described, the models of the faults which are possible in the CUT are given.
2. **INPUT PATTERN GENERATORS**
   Several ways to implement an Input Pattern Generator are presented. Their fault coverage (the number of potential faults that can be detected, divided by the total number of possible faults) is calculated, assuming a certain fault model.
3. **TEST DATA EVALUATORS**
   The outputs of the CUT must be evaluated by the Test Data Evaluator of which several implementation alternatives are presented.

3.1 FAULT MODELS

Before the interior logic test techniques are described, the faults that can occur in a VLSI circuit are discussed. The physical faults occurring in VLSI circuits will be modelled with logical faults. A physical fault depends on the design rules, manufacturing processes and chip technology (CMOS, NMOS, ECL, TTL, etc.), while a logical fault is almost technology independent and therefore more useful.

In this section four logical fault models will be discussed:

1. **Stuck-at-0 and Stuck-at-1 Fault Model**
   The stuck-at-0 and stuck-at-1 fault model is the most frequently used fault model. A fault of this type occurs when a circuit line or node remains at its logical value ('0' or '1'). The stuck-at-model is very popular, because its influence on the logical value of the circuit-line or -node is easy to understand and to implement. The stuck-at model covers about 90% of all physical faults [6].
2. **Stuck-at-open Fault Model**
   The stuck-at-open fault model represents an unintended open input of a gate. The remaining logical value of the open line depends on the used technology. If the used technology is for instance TTL, then an open line gets the logical value '1'. An open line in TTL can therefore be represented by the stuck-at-1 fault. When on the other hand the used technology is CMOS, a completely different situation occurs: the capacity and high impedance of the CMOS transistor makes the behaviour of the faulty circuit become sequential. Because these faults only occur with CMOS technology, this model is not (yet) generally used.
3. **Coupling/Bridging Fault Model**
   The bridging or coupling fault model represents the group of faults that occur when two or more points of the circuit are connected. Sometimes this kind of faults can also be modelled by the stuck-at model.
4. **Pattern Sensitive Fault Model**
   The pattern sensitive fault model represents faults that occur as a result of the change (or the impossibility to change) in a logical value due to the presence of a specific pattern of logical values at certain points in the circuit. This fault occurs mainly in memory chips. The fault is often a result of memory cells being very close together (high physical density), and occurs more often in dynamic than in static memory.

3.2 INPUT PATTERN GENERATORS

As said in the introduction of this section, the global model of the interior logic test consists of four parts, namely the Input Pattern Generator (IPG), the Circuit Under Test (CUT), the Test Data Evaluator (TDE) and the test-controller. In this subsection the IPG will be discussed.

The IPG generates patterns which result in a certain response of the CUT, observable on the output of the CUT. Three types of IPGs can be distinguished, namely:

1. EXHAUSTIVE INPUT PATTERN GENERATORS
2. DETERMINISTIC INPUT PATTERN GENERATORS
3. PSEUDORANDOM INPUT PATTERN GENERATORS

3.2.1 EXHAUSTIVE INPUT PATTERN GENERATORS

The exhaustive input pattern generator generates all possible patterns. This type of IPG can detect almost all faults in the CUT, and is therefore almost the ideal IPG. The only (big) disadvantage is the large number of patterns needed to test the circuit, namely 2^n (n is the number of input pins). When $n \geq 25$, which happens very often, the time needed to generate all patterns becomes too large. In that case the structure of the circuit must be evaluated in order to reduce the number of patterns. This can be done by dividing the circuit into several smaller parts (partitioning) ([7], [8]).

An exhaustive IPG is shown in Figure 5. It generates the following patterns (when starting with 0000):
0000, 1000, 0100, 0010, 1001, 1100, 0110, 1011, 0101, 1010, 1101, 1110, 1111, 0111, 0011, 0001, 0000, ...

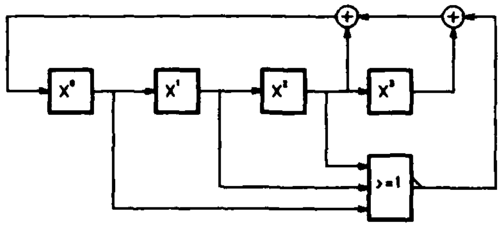


Figure 5: An exhaustive pattern generator

The exhaustive IPG of Figure 5 consists of a so-called linear feedback shift register (LFSR). This LFSR is adjusted in such way that it produces all possible patterns (including the all-zero pattern) (see also [9], [5]).

3.2.2 DETERMINISTIC INPUT PATTERN GENERATORS

The deterministic input pattern generator generates specific patterns, which were calculated during the design-phase of the circuit. The deterministic input pattern generator uses the pre-calculated patterns to detect all possible faults of the chosen fault model. These patterns can be stored in ROM on the chip (the generation is then called non-concurrent generation), or they can be generated in a "clever" way (concurrent generation) ([10]). In most cases the patterns cannot be generated concurrently, so the deterministic patterns must be stored requiring a ROM, which will increase the chip area (in most cases this area will be too large).

Three types of deterministic IPGs can be distinguished:

1. **Behavioural IPGs**
   The behavioural IPGs treat the CUT as a black box. Only the function of the CUT is known, not the way it has been implemented. When the behavioural IPG is used, it is not known which and how many faults will be detected. Therefore the behavioural IPG will only be used when the internal structure of the CUT is unknown or very difficult to understand. Therefore this type of deterministic IPG is not used as an interior logic (self) test pattern generator.
2. **Functional IPGs**
   The functional IPGs are used when a little more is known about the internal structure of the CUT (the CUT is treated as a gray box). Functional tests are particularly useful when the implementation of a circuit is known at a higher level of abstraction (for example at the data-path level) or too complex to understand in detail. They can also be used to obtain testability information during the early stages of the design phase, when the internal details of modules are not available. Since gate level models may not cover all physical failures, functional patterns are also used to check out the circuit at operating speeds. These patterns have also the advantage that invalid or illegal inputs are not applied to the circuit. Functional testing can be used in addition to other interior logic test methods in order to check specific functions. This method will not be used in those cases where the internal structure of the circuit is known, and the input patterns have to be stored. It therefore is not often used for a chip-level (built-in) self test.
3. **Structural IPGs**
   The structural IPG can be used when the complete structure, every gate of the CUT, is known. Based on a certain fault model (for instance the stuck-at fault model), the specific patterns which detect all faults can be computed. These patterns can be computed by hand or with the use of a computer. But because of the large chip area, this kind of IPG will not be used.

3.2.3 PSEUDORANDOM INPUT PATTERN GENERATORS

Pseudorandom pattern generation is often used when exhaustive testing is not possible, because of the large number of patterns and/or when the CUT does not allow deterministic testing, because of the storage required for the input patterns. A problem with pseudorandom testing is when the result of the self test is positive, one will not have a 100% certainty that the circuit is correct, because of the probabilistic nature of the test. The question with pseudorandom input pattern generators is how many patterns should be generated in order to obtain a certain probability of detecting all faults of the stuck-at fault model (e.g. 95% or 99%). An answer to this question can be obtained in two different ways:

1. **By simulating the circuit and inserting faults.** Now it is possible to determine the exact probability of not detecting a fault with a certain number of patterns (the test-length). It is also possible to calculate the test-length, if the probability that a fault is detected is given. Simulation takes much (often too much) develop- and computingtime.

2. By approximating the test-length (i.e. the number of patterns needed) and approximating the probability that faults in the circuit will not be detected. This method takes less time than simulation, but it requires the detectability profile of the circuit.

In this paper only the second way of determining the probability of not detecting a (stuck-at) fault will be discussed, because the results have been shown to be satisfactory ([11], [5]), and the first method (simulating the circuit, and all possible faults) is often not useful in practice (it would take too much time).

The following aspects of pseudorandom input pattern generators will be discussed:

1. **The Detectability Profile**
   The probability of not detecting a (stuck-at-) fault is based on the detectability profile of the circuit.
2. **Computation of the Expected Fault Coverage**
   The detectability profile is needed to calculate the expected fault coverage, which gives an impression of the number of patterns needed to test the circuit with a pseudorandom input pattern generator.
3. **Implementations of Pseudorandom Pattern Generators**
   After presenting the theory of the pseudorandom pattern generators, the way these IPGs can be implemented will be described.

The Detectability Profile

In this subsection a way to calculate the fault-characteristic of a combinational circuit needed to calculate the expected test-length will be discussed. First a definition of the detectability profile is given:

The *detectability profile* H is the number of (stuck-at-) faults that are detected by a certain number of input patterns.

The detectability profile H is expressed as a vector $[h_1, h_2, \cdots, h_N]$. Element h_k of the vector is the number of faults in the circuit that can be detected with k input patterns. The concept of the detectability profile will be explained by means of two examples.

Example 1
Consider the circuit in Figure 6. There are six possible stuck-at faults: $c_1/0$ (connection c_1 stuck-at-zero), $c_1/1$ (c_1 stuck-at-one), $c_2/0$, $c_2/1$, $c_3/0$ and $c_3/1$.

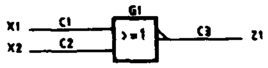


Figure 6: the test circuit, a NOR gate

There are four possible input patterns. If the input pattern is 00 ($x_1 = 0$ and $x_2 = 0$), then the normal output will be '1' ($z_1 = 1$). The fault $c_1/1$ will result in a output $z_1 = 0$, which is a wrong output. This is the only pattern which detects the fault $c_1/1$. The faults $c_1/0, c_2/0, c_2/1$ and $c_3/0$ are also detected by only one pattern (namely respectively the patterns 10, 01, 00 and 00). So there are five faults that are detected by only one pattern, therefore $h_1 = 5$. No fault is detected by only two patterns ($h_2 = 0$), one fault is detected by three patterns (namely the fault $c_3/1$, which is detected by the patterns 01, 10 and 11), and no fault is detected by four patterns ($h_4 = 0$). This results in the detectability profile $H = [h_1, h_2, h_3, h_4] = [5, 0, 1, 0]$.

□

Example 2
Compute the detectability profile H for the circuit of Figure 7.

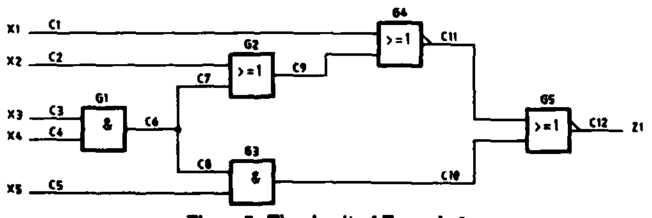


Figure 7: The circuit of Example 2

There are 24 possible (stuck-at-) faults, two for each connection c_i. The number of different input patterns is 32 (because there are 5 input-lines). For every fault the number of patterns which detect that fault has been computed, by introducing that fault and computing the value of z_1 for all possible patterns on the input lines. The number of times the value of z_1 differs from the correct value of z_1 is the number of patterns that detect that fault. This is done for all possible faults in the circuit. The result is given in Table 1.

Table 1: The number of patterns that detect a specific fault in the circuit of Figure 7

fault	number of patterns	fault	number of patterns
$c_1/0$	6	$c_7/0$	1
$c_1/1$	6	$c_7/1$	6
$c_2/0$	6	$c_8/0$	4
$c_2/1$	6	$c_8/1$	9
$c_3/0$	4	$c_9/0$	8
$c_3/1$	4	$c_9/1$	6
$c_4/0$	4	$c_{10}/0$	6
$c_4/1$	4	$c_{10}/1$	22
$c_5/0$	4	$c_{11}/0$	4
$c_5/1$	4	$c_{11}/1$	22
$c_6/0$	4	$c_{12}/0$	22
$c_6/1$	12	$c_{12}/1$	10

The detectability profile of the circuit of Figure 7 is $H = [h_1, h_2, h_3, h_4, \cdots, h_{32}]$ consists of the following elements: $h_1 = 1$ (there is one fault that is detected by only one pattern), $h_4 = 9$ (there are nine faults that are detected by four patterns), $h_6 = 7$, $h_8 = 1$, $h_9 = 1$, $h_{10} = 1$, $h_{12} = 1$, $h_{22} = 3$ and all other elements of H are zero.

□

The detectability profile of most circuits is very difficult to obtain. There are three ways of calculating the detectability profile, namely:

1. Exhaustively fault simulation. This is a brute force method, which takes (often too) much time (for large circuits). In Example 1 and Example 2 this method was used to explain the detectability profile for two simple circuits.
2. Another approach which also takes much time is the D-algorithm, which can be used to find the exact profile. This method can only be used for small circuits. [12]
3. The third method uses an approximation technique. In [5] a way to calculate an approximation of the detectability profile is described.

Computation of the Expected Fault Coverage

The *expected fault coverage* is the fraction of potential faults in the circuit that can be detected by applying a specific number of pseudorandom input patterns. The expected fault coverage can be used to get insight in the number of patterns needed to obtain a certain probability of detecting all faults. Before the expected fault coverage can be calculated, the detectability of a fault must be defined:

The *detectability* k, of a fault is the number of patterns that will detect that fault.

In the derivation of the equation of the expected fault coverage the following symbols are used:

n	is the number of input lines of the circuit.
N	is the number of different possible patterns, $N = 2^n$.
M	is the total number of possible faults.

In [13] and [14] the equation for the expected fault coverage $E(C_L)$ is derived:

$$E(C_L) = 1 - \frac{1}{M} \sum_{k=1}^{N-L} \frac{h_k}{\binom{N}{k}} \binom{N-L}{k}, \tag{1}$$

where L is the number of patterns that have been applied, and h_k the k^{th} element of the detectability profile vector H.

It can be shown that the expected fault coverage $E(C_L)$ is small for circuits with faults with a small detectability k, i.e. with faults which are only detected by a *few* (k) test-patterns.

Example 3 shows the use of Eq.1.

Example 3
The detectability profile of the circuit of Figure 7 (see Example 2) is $H = [h_1, h_2, h_3, \cdots, h_{32}]$, with $h_1 = 1$, $h_4 = 9$, $h_6 = 7$, $h_8 = 1$, $h_9 = 1$, $h_{10} = 1$, $h_{12} = 1$ and $h_{22} = 3$. The total number of different input patterns is 32 (N=32) and there are 24 different possible faults (M=24). Now the expected fault coverage for different test lengths L can be calculated with Eq.1. The result is shown in Table 2.

Table 2: The expected fault coverage $E(C_L)$ for the circuit in Example 2 for different test-lengths L

L	$E(C_L)$	L	$E(C_L)$
1	0.239	12	0.910
2	0.392	13	0.925
3	0.504	14	0.938
4	0.591	15	0.948
5	0.661	16	0.957
6	0.720	17	0.964
7	0.768	18	0.970
8	0.807	19	0.975
9	0.841	20	0.978
10	0.868	25	0.990
11	0.891	32	1.000

For a fault coverage of 95% 16 input patterns are needed. Figure 8 shows how the expected fault coverage increases when the number of patterns L increases.

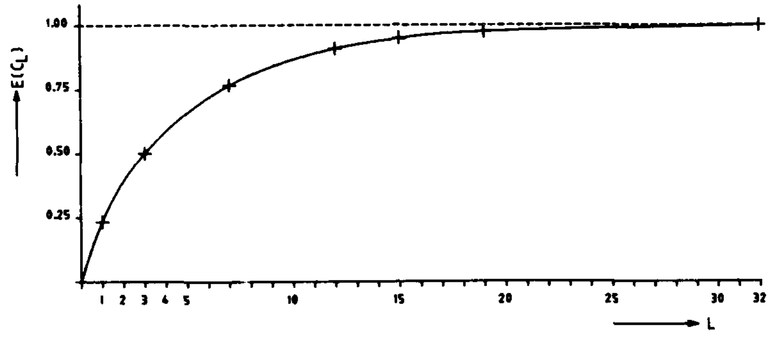


Figure 8: The expected fault coverage $E(C_L)$ as a function of the test-length L

□

Because of the probabilistic approach of Eq.1, the expected fault coverage can differ from the real fault coverage (which can be computed through (fault-) simulation). Wagner et al. [11] shows that in most cases the expected fault coverage differs less than 2.5 percent from the real value.

Implementations of Pseudorandom Pattern Generators

The pseudorandom pattern generator (PRPG) is mostly implemented using a LFSR (Linear Feedback Shift Register). A LFSR consists of a string of registers (flip-flops/latches) and (exclusive-) OR-gates. Several output lines of the registers are fed back to the input. The number of registers and the way the feedback paths are implemented, together with the *seed* (the initial contents of the registers) determines the length of the test sequence before it repeats. The length of a test sequence (the number of different patterns that are generated) can be at most 2^n (when the LFSR has n registers).

In this section three pseudorandom pattern generators are given, namely a general PRPG, a so-called "maximum length" PRPG and a "maximum-length" PRPG including the all-zero pattern.

Figure 9 shows the implementation of a LFSR with three registers and three feedback paths (using two XOR-gates). These feedback paths determine the so-called "characteristic polynomial" or "function" of the LFSR. The characteristic polynomial of the LFSR of Figure 9 is x^3+x^2+x+1. The way the characteristic polynomial of a LFSR can be computed is given in [9], [5] and [15].

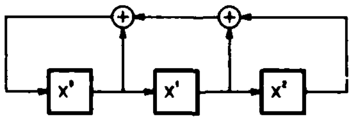


Figure 9: A general LFSR with characteristic polynomial x^3+x^2+x+1

When the initial contents of the registers of Figure 9 is 000, then the contents of the registers of the LFSR (after a clock-pulse) will also be 000. So the number of different patterns produced with seed 000 is 1. When the seed is 100, four patterns are generated before the sequence repeats, namely 100, 110, 011, 001, 100, ...

Now consider the LFSR of Figure 10.

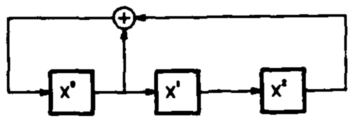


Figure 10: A "maximum length" LFSR with characteristic polynomial $x^3 + x + 1$

When the seed of the LFSR is 100, the number of different patterns that are generated will be 7 (the maximum number of patterns for a LFSR with three registers), namely: 100, 110, 111, 011, 101, 010, 001 , 100, ... The characteristic polynomial of the LFSR is $x^3 + x + 1$.

The question now is how to make a "maximum length" LFSR? Wang et al. [15] shows that for a LFSR to have maximum length, the characteristic polynomial should be a so-called "primitive" polynomial. Whether a polynomial $P(x)$ is primitive can be determined by dividing a polynomial $x^m + 1$ by the (possible) primitive polynomial $P(x)$. When the smallest integer m is equal to $2^n - 1$ (when n is the number of registers of $P(x)$), then the polynomial $P(x)$ is primitive (see also [5]).

The main disadvantage of a normal "maximum length" pseudorandom pattern generator is that it cannot generate the all-zero pattern. The problem of not being able to produce the all-zero pattern can be solved by adding a NOR-gate and an XOR-gate (see [5]). Figure 11 shows the LFSR of Figure 10 changed in such way that it produces all 2^n patterns (namely the patterns 100, 110, 111, 011, 101, 010, 001, 000, 100, ...).

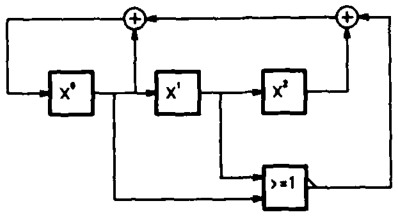


Figure 11: LFSR with characteristic polynomial $x^3 + x + 1$ with the all-zero pattern

The inclusion of the all-zero pattern costs a (large) NOR-gate and one XOR-gate. Although it has been stated that a PRPG should generate all patterns with equal probability in order to calculate the expected fault coverage, usually it will not be necessary to generate the all-zero pattern. It is obvious, that when the total number of possible patterns is large in comparison with the test-length, the probability of generating the all-zero pattern is very small and therefore the hardware to generate the all-zero pattern will not be worthwhile. It will only be useful when (through simulation) it can be shown that certain faults can only be detected with the all-zero state.

3.3 TEST DATA EVALUATORS

This section discusses the implementation alternatives for the Test Data Evaluators (TDEs). The function of the TDE is to determine whether the response of the CUT to the input data is correct. A global model of the TDE is shown in Figure 12.

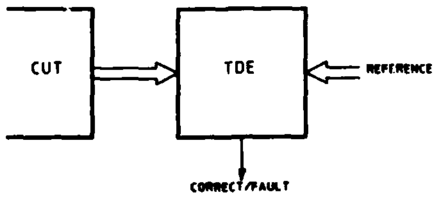


Figure 12: Global model of a TDE

The TDE must compare the data from the CUT with so-called reference-data (the data that is known to be correct). This can be done in two ways: with or without a compressor. (see Figure 13(a) and (b)).

The problem with non-compressing TDEs is that they need a large storage capacity for the reference data. Therefore the non-compressing TDE is seldom used. In this paper only compressing TDEs will be discussed.

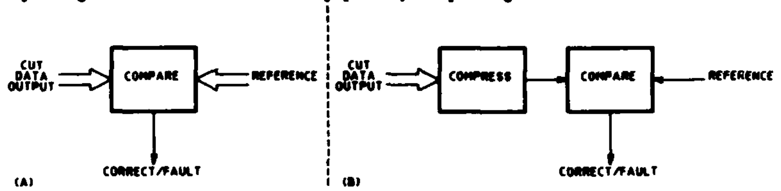


Figure 13: (a) Non-compressing TDE. (b) Compressing TDE

The following subjects will be described:

1. **OVERVIEW OF COMPRESSING TECHNIQUES**
   The major compressing techniques are discussed.
2. **CRC CODING THEORY**
   The theory of the most commonly used compressing technique (using CRC coding) is described.
3. **IMPLEMENTATIONS OF CRC SIGNATURE ANALYZERS**
   The way CRC signature analysers can be implemented are described.
4. **MASKING-CHARACTERISTICS OF CRC SIGNATURE ANALYZERS**
   The types of faults not detected (masked) with the CRC signature analyser are looked at.

The section ends with conclusions on the subject of test data evaluators.

3.3.1 OVERVIEW OF COMPRESSING TECHNIQUES

In almost all cases the TDE consists of a data compressor and a comparator (see Figure 13(b)). The data compressor maps a long stream of data from the CUT, into a relatively short word, called a signature. An ideal data compressor would have the property that a faulty input data stream will always yield a faulty signature. However, no such data compressor is in existence, because high compression rates can only be achieved with non loss-free compression techniques, i.e. techniques which cannot reproduce the original data *exactly*. The action of mapping an erroneous data stream into a correct signature is called *error masking* or *aliasing*. The best one would hope to get from a data compressor is a small probability of error masking, which expresses the quality of a data compressor.

The major ways of compressing data are:

1. **Parity Checking**
   A TDE using the parity checking method checks the parity of the data output stream of the CUT. A parity check will detect any faults produced by an odd number of error bits, so all single bit errors will be detected. Faults due to an even number of error bits will not be detected. Therefore this compression method has an error masking probability of about 0.5, which is not acceptable.
2. **Ones Counting**
   In ones counting, as the name suggests, the data compressor is merely a counter, counting the number of logical ones in the data output stream of the CUT. It is a ${}^2\log t$-bit counter (where t is the number of data output bits of the CUT). Ones counting will detect any single bit error and a large part of the multiple bit errors (see [16]).
3. **Transition Counting**
   Transition counting counts the number of times the output data stream changes value (from $0 \rightarrow 1$ or from $1 \rightarrow 0$). As with ones counting, the count length is proportional to ${}^2\log$ of the length of the data output stream of the CUT. It can be shown that a transition counter which is counting large data streams, will have an error masking probability of 0.5. Therefore this method is seldom used [1].
4. **Cyclic Redundancy Check Coding**
   The cyclic redundancy check (CRC) technique uses a linear feedback shift register to compress the data of the CUT. The contents of the register depends on data output stream of the CUT and the previous contents of the register. The contents that remains in the register is the *signature* and will be compared with reference data. Compressors using CRC coding are very popular, because of their simple implementation and good (small) masking probability. The theory of the CRC compressing technique will be described more detailed in Section 3.3.2.

3.3.2 CRC CODING THEORY

The main purpose of a signature analyser is to compress the data coming from the CUT to a code word, called *signature*. The compression is based on the CRC coding theory. The CRC coding theory treats streams of binary data as polynomials with binary coefficients. A binary data stream of t-bits ($p_0p_1p_2\cdots p_{t-2}p_{t-1}$, where p_i is 0 or 1) can be represented as a polynomial with t terms, namely the polynomial $p_{t-1}x^{t-1} + p_{t-2}x^{t-2} + \cdots + p_1x + p_0$ (see Example 4). The *degree of* the polynomial is the largest power of x in a term of the polynomial with a non-zero coefficient. Only the coefficients are of interest, not the actual value of x.

Example 4
The input-stream 1001011 can be represented by the polynomial $x^6 + x^5 + x^3 + 1$. The degree of this polynomial is 6.

□

One can perform arithmetic on the polynomials, with coefficient addition and multiplication modulo-2. Hence there are no carries during addition, and when binary polynomials are added, multiplied or divided, the result is also a binary polynomial (see [9] [5]). Division of one polynomial by another produces a quotient polynomial $Q(x)$ and a remainder polynomial $R(x)$.

$$\frac{P(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$

The remainder $R(x)$ is often used as the "signature" of the data output stream from the CUT.

3.3.3 IMPLEMENTATIONS OF CRC SIGNATURE ANALYZERS

In this section implementations of signature analyzers using the CRC coding method will be discussed. There are generally two types of CRC signature analyzers, namely:

1. **"One-input" Signature Analyzer: LFSR**
   When only one output of the CUT is observed, a LFSR (Linear Feedback Shift Register) can be used as CRC signature analyzer.
2. **"Multi-input" Signature Analyzer: MISR**
   The MISR (Multi Input Signature Register) observes multiple output lines of the CUT at the same time.

"One-input" Signature Analyzer: LFSR

A LFSR (Linear Feedback Shift Register) consists of three parts, namely the register (consisting of flip-flops / latches), exclusive-OR gates and feedback paths. The way the three parts of the LFSR are configured determines the behaviour of the LFSR. Two of those variations will be described:

1. **LFSR with Distributed Feedback**
   A LFSR with distributed feedback is shown in Figure 14.

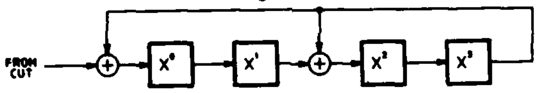


Figure 14: LFSR with distributed feedback and divisor polynomial $x^4 + x^2 + 1$

The number of flip-flops and the way the feedback lines are connected, determines the so-called divisor polynomial, which is the characteristic of the LFSR. The *quotient* of a LFSR are the bits (the data) shifted out of the LFSR and the *remainder* of a LFSR is the contents of the registers after the shift operations (see Example 5).
The divisor polynomial $G(x)$ is determined by the number of flip-flops and the way the feedback lines are connected (see [9] [5]).

Example 5
If an input stream 110001001 ($P(x) = x^8 + x^5 + x + 1$) is fed into the LFSR input of Figure 14, then the LFSR comes into the states of Table 3.

Table 3: LFSR states during a division

input	x^0	x^1	x^2	x^3	comment
	0	0	0	0	initial state
1	1	0	0	0	
0	0	1	0	0	
0	0	0	1	0	
1	1	0	0	1	
0	1	1	1	0	
0	0	1	1	1	
0	1	0	0	1	
1	0	1	1	0	
1	1	0	1	1	$R(x) = x^3 + x^2 + 1$

The divisor polynomial is $G(x) = x^4 + x^2 + 1$. The quotient $Q(x)$ (011010000 is $x^4 + x^2 + x$) is shifted out of the LFSR and $R(x)$ (the remainder, or $x^3 + x^2 + 1$) remains in the register after the 9 clock pulses.

□

This LFSR structure will create a "code word" ($R(x)$) as the remainder after dividing the message polynomial ($P(x)$) by the feedback polynomial ($G(x)$). $R(x)$ is called the *signature* of the data stream $P(x)$. Therefore the LFSR is called a *signature analyzer*.

2. **LFSR with Centralized Feedback**
The LFSR of Figure 14 needed XOR gates between several stages of the shift register. The alternative implementation (see Figure 15) does not have those gates, but has only one XOR gate, at the input of the LFSR.

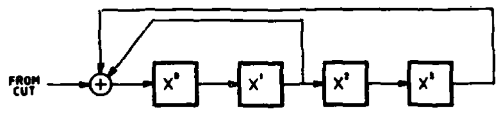


Figure 15: LFSR with centralised feedback and divisor polynomial $x^4 + x^2 + 1$

The "remainder" (the value of the registers after the self test) is not the remainder of a division (as with the distributed LFSR), but has the same masking-characteristics as the signature of a LFSR with distributed feedback. This implementation can therefore also be used to generate a signature of an input data stream ([17]).

"Multi-input Signature Analyzer: MISR

For testing a CUT with multiple outputs, the circuitry overhead of a single input signature analyzer on every output would be high. The single input analyzer could be multiplexed to the various outputs of the CUT, one at a time, and repeat the test sequence for each output. However, a parallel signature testing method using a structure called a Multiple Input Signature Register (MISR) (see Figure 16) is preferred, because the compression is done in parallel for all outputs of the CUT.

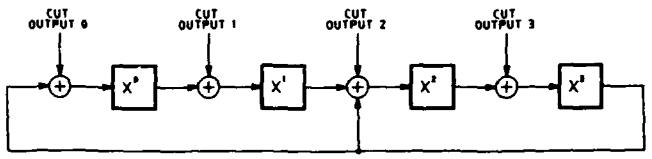


Figure 16: A multiple input signature register

Again the value of the registers after the self test is called the signature or the remainder of the MISR.

3.3.4 MASKING-CHARACTERISTICS OF CRC SIGNATURE ANALYZERS

The main concern with the use of signature analysers using the CRC coding technique is the probability that an erroneous stream from a faulty CUT could be compressed into the same signature as a good CUT. This is called masking or aliasing. In the next sections the masking characteristics of the described implementation of a signature analyzer with a LFSR and with a MISR will be described.

Masking-characteristics of LFSRs

According to [17], a signature analyzer with the LFSR with centralized feedback has the same masking characteristics as a LSFR with distributed feedback.

A fault in a CUT can cause an error on the CUT output in one or more bit positions of the output stream. The possible error patterns can be treated as polynomials, where each non-zero coefficient in the error polynomial $E(x)$ represents an error in the corresponding bit position of the CUT output stream.

Example 6
If the nineth, third and second bit of an output stream is erroneous, the error polynomial $E(x)$ is $E(x) = x^8+x^2+x$.

□

$P_E(x)$ (which is the actual output stream of the faulty CUT) is: $P_E(x) = P(x) + E(x)$, where $P(x)$ is the output of a correct CUT.

Example 7
If the correct CUT data output $P(x)$ is $P(x) = x^8+x^6+x+1$ and the error polynomial $E(x)$ is $E(x) = x^8+x^2+x$ then the result (the incorrect CUT data output) will be $P_E(x) = x^6 + x^2 + 1$.

□

As $P(x) = Q(x) * G(x) + R(x)$ and $P_E(x) = P(x) + E(x)$, then an undetectable error will occur if $P_E(x)/G(x)$ will have the same remainder $R(x)$ as $P(x)/G(x)$. This means that $P_E(x) = Q_E(x) * G(x) + R(x)$.

This will be illustrated with an example:

Example 8
If the divisor polynomial of the LFSR is $G(x) = x^4 + x^2 + 1$ and the correct CUT output stream is $P(x) = x^8 + x^5 + x + 1$, then the remainder will be $R(x) = x^3 + x^2 + 1$ and $Q(x)$ (the quotient polynomial of the correct CUT) will be $Q(x) = x^4+x^2+1$. But if the error polynomial $E(x)$ of the faulty CUT is $E(x) = x^8+x^5+x^4+x^3+x+1$ then $P_E(x)$ (the faulty CUT output stream) is $P_E(x) = x^4 + x^3$. This results is the remainder $R_E(x) = x^3 + x^2 + 1$, which is the same polynomial as the one that remains in the LFSR after a correct CUT output stream has been applied.

□

If the degree of the correct data polynomial is $t - 1$ (the data stream has t-bits), then there are $2^t - 1$ possible error polynomials, because an error can be represented by a polynomial of degree $t - 1$ or less (the error polynomial $E(x) = 0$ is eliminated). If $G(x)$ is of degree m, then it has $2^{(t-m)} - 1$ non-zero multiples of degree less than t. So there are $2^{(t-m)} - 1$ wrong t-bit streams that can map into the same signature as the correct data stream. If all error polynomials are equally likely, then the probability that the divisor of degree m will not detect an error is

$$P_{masking} = \frac{2^{(t-m)} - 1}{2^t - 1}$$

and for large t it means that

$$P_{masking} = \frac{1}{2^m} \qquad (2)$$

so the "masking-probability" can be made quite small by increasing the degree m of the divisor polynomial $G(x)$.

Note that Eq.2 is only correct when the error polynomials are equally likely.

Masking-characteristics of MISRs

It can be shown that a MISR has masking characteristics which are quite similar to those of a single input signature analyser (LFSR) ([1], [5]).

If all possible error polynomials are equally likely then the probability that a MISR using $G(x)$ as the divisor will not detect an error is

$$P_{masking} = \frac{2^{(t-1)} - 1}{2^{(m+t-1)} - 1} \approx \frac{1}{2^m} \qquad (\text{if t = large}),$$

(with t is the number of data bits that are collected (or the number of clock-pulses), and m is the number of registers of the MISR). So the "masking-probability" can be made quite small by increasing the degree m of the divisor polynomial $G(x)$.

3.3.5 DIVISOR POLYNOMIAL CONSIDERATIONS

The usefulness of CRC signature analysers has been shown above. There remains still one question: what divisor polynomial should be used for signature analysis? The masking probability $P_{masking} \approx 2^{-m}$ for equally likely error patterns and long data streams. This implies that m (the degree of the divisor polynomial) should be large enough. Some errors can be distinguished (for a detailed analysis, see [5]), namely:

1. **Single Bit Errors**
   Single bit errors are those errors that affect only one bit in the output stream. These errors will be detected by any polynomial with two or more non-zero coefficients.
2. **Burst Errors**
   Burst errors are those errors that occur when within a d-bit output stream, at most d bits are erroneous. These errors will be detected by a signature analyser with polynomial $x^m + 1$, where $m \geq d$.
3. **Single Output Errors**
   Single output errors occur when only one input of the CUT gives wrong values (and when the outputs of the CUT are connected with a MISR). The probability that an single output error will be masked is the same as the masking probability for LFSRs.
4. **Error Cancellation Errors**
   The only kind of errors that can occur in MISRs which will result in a higher masking probability are the error cancellation errors. But it can be shown that these errors will not have a significant part in the complete masking probability, so these kind of errors are negligible.

But still the fact remains that a divisor polynomial $G(x)$ will have a certain probability that it will not detect an error polynomial $E(x)$ which is a multiple of $G(x)$. One thought to deal with this problem is to incorporate a rather complex feedback structure into the LFSR (MISR), on the assumption that a complex $G(x)$ will prevent masking by anything but an equally complex $E(x)$. The Hewlett-Packard 5004A Signature Analyser [18] for example uses a 16-stage LFSR with $G(x)$:

$$G(x) = x^{16} + x^9 + x^7 + x^4 + 1.$$

Another example of polynomials which are used in the field is the CCITT-16 polynomial

$$G(x) = x^{16} + x^{12} + x^5 + 1,$$

which is used in microprocessors (the MC68020 [19] and the MC6804P2 [20]) or co-processors (the MC68881 floating point processor [21]).

Therefore the only "answer" one can get is that $G(x)$ should be chosen in such way that no expected fault in the CUT will create a polynomial that is a multiple of $G(x)$. A polynomial of a high degree and a complex feedback structure is an accepted practical rule. By simulating the circuit the polynomial can be proved to be satisfiable.

4 BOUNDARY SCAN TEST

The third phase of the self test of a circuit (the boundary scan test) tests the so-called exterior logic. Exterior logic is combinational logic, which is at one side connected with logic from another circuit, and at the other side it is connected with the (internal) registers of the circuit (see Figure 17). Since the exterior logic is difficult to test, the exterior logic should be held as small as possible.

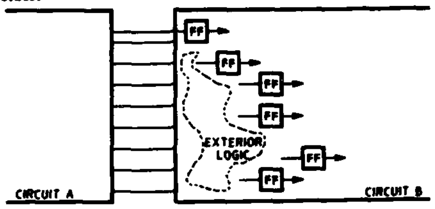


Figure 17: The exterior logic

When a scan-path test (for the registers) and a internal BIST (for the internal logic) has been applied, the only combinational logic that remains untested is the exterior logic. The problem of testing exterior logic has been recognised by some large companies (mostly large scale users or producers of printed circuit assemblies and systems), which have formed the Joint Test Action Group (JTAG) group is formed to deal with this problem.

The solution to the problem of not testing the exterior logic is, according to JTAG, called boundary scan test ([22]). The objective of boundary scan test is to separate the exterior logic from the other logic of the circuit. Patterns can be applied at the input signals of the exterior logic, and the results can be collected at the output signals of the exterior logic. This is done by configuring the registers (flip-flops, latches), which are connected to the exterior logic, such, that during the boundary scan test the contents of these registers can be controlled by the boundary scan test controller (which is not part of the circuit).

The question that arises now, is how should the input and output registers be re-configured such that they can act like a shift register during boundary scan test phase, and that they are able to collect the data on the input lines.

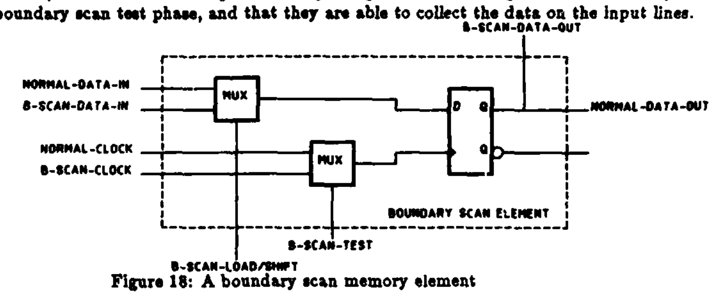


Figure 18: A boundary scan memory element

The re-configuration is drawn in Figure 18. There are five extra control and data lines:

1. **b-scan-data-in**
   Through this data line the data coming from the previous boundary scan memory element (from "b-scan-data-out") (or the boundary scan test controller) is given to the boundary scan element.
2. **b-scan-data-out**
   This data line is used to shift the data out the boundary scan memory element, to the next boundary scan element (or the boundary scan test-controller).
3. **b-scan-load/shift**
   This control line determines whether during the boundary scan test the data coming on the "b-scan-data-in" line, or the data coming from "normal-data-in" is clocked in the register. During normal operation this line should indicate that the data on the "normal-data-in" line should be clocked in the register.
4. **b-scan-test**
   This control line determines whether it is a boundary scan test, or a normal operation.
5. **b-scan-clock**
   This line is used to clock the data in the register during boundary scan test.

5 CONCLUSION

In this paper the possibilities of Built-In Self Test have been examined. For different kinds of logic, different types of self tests have been described. For the registers the self test is called scan-path test. The interior logic is tested with one of the described interior logic tests. The exterior logic is tested with the boundary scan test method. So each test takes care of of a specific part of the circuit, and all these tests together cover the complete circuit. After the complete self test, all logic has been tested, except the logic that controls the clock. The extra test-control hardware (see Figure 4) needed to perform a self test can be combined for the several self test techniques. In that case one can design a special BIST-scan element, which replaces the registers in the circuit.

References

[1] A.J. van de Goor, *"Digital System Testing, Hardware, Software and System requirements"*, Delft University of Technology, Delft, 1987.

[2] R.J. Illman, "Design of Self Testing RAM", *Silicon Design Conference 1986*, pp. 439-446, 1986.

[3] S.K. Jain and C.E. Stroud, "Built-in Self Testing of Embedded Memories", *IEEE Design and Test of Computers*, Vol. 3, no. 5, pp. 27-37, October 1986.

[4] T. Sridhar, "A New Parallel Test Approach for Large Memories", *IEEE Design and Test of Computers*, Vol. 3, no. 4, pp. 15-22, August 1986.

[5] B. Jansen, *Built-In Self Test of the Image Memory System*, Master Thesis report, Delft University of Technology, Faculty of Electrical Engineering, 1987.

[6] T.W. Williams, "VLSI Testing", *Computer*, Vol. 17, no. 10, pp. 126-127, October 1984.

[7] L-T. Wang and E.J. McCluskey, "Circuits for Pseudo-Exhaustive Pattern Generation", *Proceedings 1986 International Test Conference*, pp. 25-37, 1986.

[8] E.J. McCluskey and S. Bozorgui-Nesbat, "Design for Autonomous Test", *IEEE Transactions on Computers*, Vol. C-30, no. 11, pp. 866-875, November 1981.

[9] P.H. Bardell, J.A. Abraham, W.H. McAnney and J. Savir, *"Built-In Test: Theory and Implementations"*, a Tutorial on Built-In Testing, 1985.

[10] W. Daehn and J. Mucha, "Hardware Test Pattern Generation for Built-In Testing", *Proceedings 1981 International Test Conference*, pp. 110-113, 1981.

[11] K.D. Wagner, C.K. Chin and E.J. McCluskey, "Pseudorandom Testing", *IEEE Transactions on Computers*, Vol. C-36, no. 3, pp. 332-343, March 1987.

[12] J.P. Roth, "Diagnosis of automata Failures: A Calculus and a Method", *IBM Journal of Research and Development*, Vol. 10, pp. 278-291, July 1966.

[13] Y.K. Malaiya and S. Yang, "The Coverage Problem for Random Testing", *Proceedings 1984 International Test Conference*, pp. 237-245, 1984.

[14] C.K. Chin and E.J. McCluskey, "Test Length for Pseudorandom Testing", *IEEE Transactions on Computers*, Vol. C-36, no. 2, pp. 252-256, February 1987.

[15] L-T. Wang and E.J. McCluskey, "A Hybrid Design of Maximum Length Sequence Generators", *Proceedings 1986 International Test Conference*, pp. 38-47, 1986.

[16] J.Losq, "Efficiency of Random Compact Testing", *IEEE Transactions on Computers*, Vol. C-27, no. 6, pp. 516-525, June 1978.

[17] J.E. Smith, "Measures of the Effectiveness of Fault Signature Analysis", *IEEE Transactions on Computers*, Vol. C-29, no. 6, pp. 510-514, June 1980.

[18] R.A. Frohwerk, "Signature Analysis: A New Digital Field Service method", *Hewlett-Packard Journal*, Vol. 28, no. 5, pp. 2-8, May 1977.

[19] J. Kuban and J. Salick, "Testability Features of the MC68020", *Proceedings 1984 International Test Conference*, pp. 821-826, 1984.

[20] W.C. Bruce and J.R. Kuban, "Self-Testing the Motorola MC6804P2", *IEEE Design and Test of Computers*, Vol. 1, no. 2, pp. 33-41, May 1984.

[21] L.A. Basto and J.R. Kuban, "Test Features of the MC68881 Floating Point Coprocessor", *Proceedings 1985 International Test Conference*, pp. 752-757, 1985.

[22] F.P.M. Beenker, C.M. Maunder and C. Vivier, *"A Standard Boundary Scan Architecture"*, 1986.

ACKNOWLEDGEMENTS

The authors would like to express their thanks to ir. J.H.M. Deckers (Philips Medical Systems), who supported and stimulated this research. They also want to thank Océ Nederland BV for giving the opportunity to proceed the research, and present this paper.

DISCUSSION

M.Jacobsen, Ge
Have you actually implemented your BIT and if so what results are available?

Author's Reply
The research on the subject Built-In Self Test was only a literature study; there was no opportunity to actually design a circuit (chip) with BIST. But in practice there are some chips which have a BIST facility (for instance Motorola 68020) and they claim to cover many of the possible faults with their BIST circuit.

W.Mansel, Ge
Can the BIST build-in-self-test methods be applied for standard microprocessors?

Author's Reply
The BIST methods can only be used when they are implemented inside the chip, so a special chip must be designed.

R.Ardrey, US
Can BIST be initiated at any time during circuit activity with retention of state or only at system initialization (power-up)?

Author's Reply
BIST can only be used at system initialization.

AVIONICS EXPERT SYSTEMS

Dr. Ulrich D. Holzbaur
AEG Radar Subdivision, Data Processing
Airborne Radar System Group
Sedanstr.10, Post Office Box 1730
D-7900 Ulm, West Germany

0. Summary

We present some experiences and new ideas on the expert systems (XS) approach to solving problems of avionics data processing. We want to focus on problems concerning fighter aircraft and within these mainly on radar components.

Avionics expert systems may be a development tool, part of a ground support system or integrated in a avionics dataprocessing system. In the aircraft, main applications comprise mission oriented tasks like pilot's associate and sensor oriented tasks like navigation or radar.

Areas in which more intelligent radar data processing is needed comprise situation awareness, mode selection and electronic counter counter measures (ECCM), noncooperative target identification (NCI) and radiation management.

To show the applicability of the expert systems approach to identification tasks, a demonstration system for the recognition of aircrafts from the textual description of a picture was developed and tested.

1. Introduction

1.1. Expert Systems

Before discussing Avionics XS, we should state what an expert system is and when a software system does deserve the name of an expert system. Various definitions are prefered depending on the (local and disciplinary) origin of the author so, e.g. criteria for XS involve: the problem solving skill and performance of an expert; knowledge of one or several experts; inferential thinking; knowledge and explicit problem solving methods; and the system components and capabilities listed below.

We prefer to say that an knowledge based system is a computer system that contains knowledge about the problem area in explicit form and has the ability to combine this knowledge in order to solve problem, and that an expert system is a knowledge based system that contains facts and problem solving knowledge of one or several experts.

In the following, we list the essential and characteristic parts of an expert system:

- knowledge base
  - knowledge representation
    - taxonomy, structural properties
    - facts, rules
  - problem solving knowledge
    - metarules
    - metaknowledge (knowledge about knowledge)
    - control structures
- inference engine
  - question solver
  - truth maintenance system
- user interface
  - user friendly interface
    - mouse, windowing, menues
    - help facilities and explanations
  - user model
- knowledge acquisition component
  - knowledge editor
  - truth maintenance system
  - learning component
    - fact learning
    - structure learning

None of these components must be seen independently, so e.g. a truth maintenance system is on the first look part of the knowledge acquisition component, but it needs an inference engine and meta knowledge and it is essential for a learning component, which may itself be seen as part of the inference engine.

1.2. Avionics Expert Systems

Application of expert systems technology to avionics software development has the main goals to make avionics data processing systems more "intelligent" and to make the process of software production more efficient.

Main steps on the way towards "intelligent" systems are : smarter avionics systems and sensors, clever reaction to a hostile environment and gracefull degradation in case of failures for the whole system and an optimal support for the pilot and weapons system operator. This implies the ability to process data from different sources that are uncertain, noisy, inconsistent, and sometimes contradictory.

The aim of more efficient software production is achieved by the process of explicit knowledge engineering and modelling, rapid prototyping with its iterative cycle of test and improvement and by the use of AI tools and methods such as system-browsers and object-oriented programming.

As with most other expert systems, avionics applications are mostly dedicated to solving routine problems and make routine decisions leaving the final decision and exception handling to the man in the loop.

1.3. Expert Systems and Related Disciplines

1.3.1. Artificial Intelligence

XS are mostly seen as part of the AI technology. In fact, learning and more intelligent systems are reached by AI-techniques. But we think that AI as a science is more related to analysing or simulating thought processes with the help of computers than to the solution of real-world problems. Expert systems is more a technical discipline defined by its methodology. Artificial Intelligence can be seen as the field of studying problem solving strategies that can be applied when not enough problem solving knowledge about a problem is available to employ more powerful problem solving techniques (weak methods, Newell,1969). With this definition, Expert Systems are located somewhere between the Artificial Intelligence and the classical decision support or operations research approaches. Although there is no powerful analytical approach (such as e.g. control theory), the experts problem solving skill and knowledge provides a proven and effective solution technique.

1.3.2. Conventional Software Systems

The essential difference of expert systems to convenional software are the facts that the knowledge - facts and problem solving methods - is stated explicitely and not hidden as part of formulae, control structures, data and data structure and that this knowledge is passed to the system by the expert and is filtered by the programmer or knowledge engineer only with regard to structures.

Any data processing system is more or less intended to make routine decisions or to support human decisions. Decision support becomes more and more important as the software comes closer to human abilities and to the human interface, hence is closely related to the XS tasks.

1.3.3. Software and Knowledge Engineering

The process of modelling and structuring of the problem and its solution is a central feature of the design of systems that work in such complex environments and is therefore essential in the design of XS. The emphasis in building expert systems is much more on modelling the real world problem and the solution process than on data and control and their resp. flows. One may say that XS tools provide a way to "implement by specification". Going directly from the model problem to a high level solution bypasses the data processing details, hence concentrates on a higher lever of problem solving. The difference to the programming implementation is comparable with the leap between the HOL implementation and assembly language progamming.

On the other side, a rapid-prototyped XS itself may be seen as a model for some SW system. It is a well established engineering tradition to make small models in order to test great systems one intends to build. Mechanical models have been build for ships, dams, aircraft and many other objects to assess their aerodymanical, mechanical and optical properties. XS tools give the ability of rapid prototyping. Hence, one can make models of the final SW in order to test the behaviour and user acceptance of the final SW with a small effort compared to that one needed to implement the final SW. Where the time and size constraints do not allow to use an expert system for a special application, the knowledge base together with the inference engine may serve as a specification of the software that has to be implemented. Solving a problem with a high level system also provides a deeper insight into the concepts and connections inherent to the problem which may be transferred to algorithms and data structures in later stages of the development.

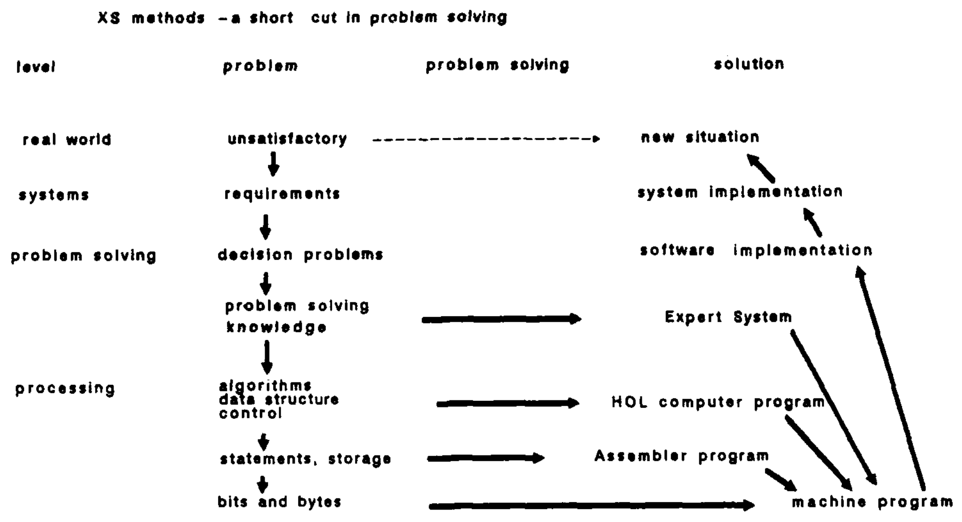


Figure 1.: Expert systems provide a direct way of implementation by specification.

2. Application Areas for Avionics Expert Systems

In the following we give an overview on those areas of application in which XS may be used as part of a system. Applications for avionics expert systems can be roughly divided into two areas: airborne XS and ground support.

Among the ground support XS there are many applications that are not specific to avionics: simulation, design, education, logistics, configuration, diagnosis, maintenance and error recovery. We leave also aside all XS applications as a support for conventional SW development such as configuration management and AI tasks such as automatic programming. Even with these restrictions there remain a lot of special applications such as mission briefing or library generation.

The airborne systems may be divided into two categories, following (McTigue, 1982): mission oriented and sensor oriented. Among the mission oriented tasks are A/A and A/G steering and weapon launch/release computations, sensor fusion, selection of best available data, integrated display management and warning functions. Mission oriented tasks concern the aircraft and its weapons or the pilot and navigator. The sensor oriented task concern single sources of information like communication, radar, inertial navigation, air data, and single displays. The tasks differ widely but they have in common such tasks as source and filter selection, error estimate and multi-(sub)-sensor fusion, failure detection and the detection of spoofing or jamming.

Graceful degradation is a point that is not special to a single system but is (or should be) an essential attribute of any AI system. Fallback solutions may be implemented explicitely when the expert takes into consideration the possible missing information or failure modes, or implicitly when the inference engine uses the knowledge about the system states to provide the best results from the incomplete and contradictory inputs.

We leave aside all typical AI tasks such as pattern and speech recognition and understanding. In the following, avionics applications for expert systems will be ordered in 3 groups with a growing difficulty of realisation: Decision support and surveillance, autonomous sensor oriented decisions and mission oriented decisions. An additional last group will scetch some ground based applications.

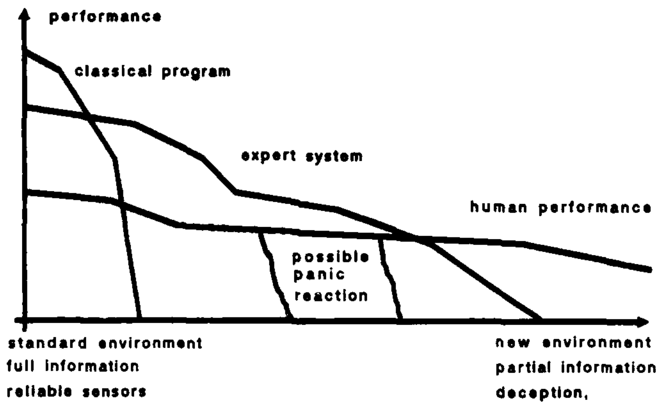


Figure 2.: Intelligent systems provide graceful degradation and adaption to changes in environment and to lack of information.

2.1. Decision Support Near the Human Interface

Decision support, display control and surveillance are among the most promising areas for avionics expert systems for the next few years. We have apt human experts that can give their knowledge to support newcomers and to take much routine load from the skilled colleagues. In this area, the amount of incoming data and reaction time has to lie in the order of magnitude of that for human data processing. The following tasks also concern radar systems but are not radar specific.

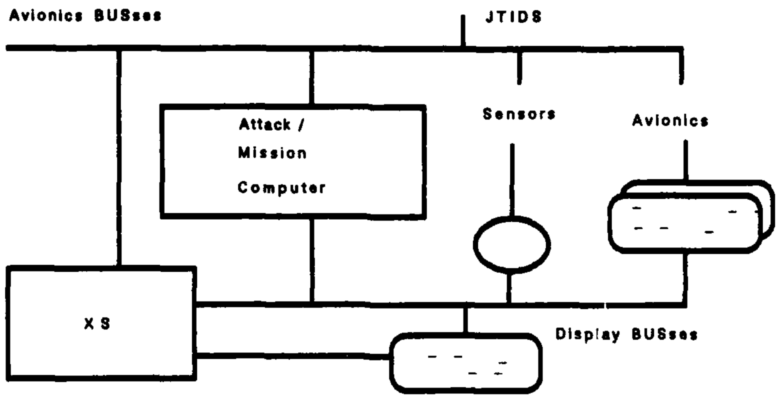


Figure 3.: Possible location for a display and decision oriented Avionics XS

2.1.1. Mode and Display Control

Mode selection for displays is based on a lot of experience. To select the contents and form of a display depending on the whole situation needs the assessment of the situation and e.g. consciousness of pilot's concentration. Graceful degradation is provided by bridging gaps in the incoming data and decluttering the display by thoroughly weighting the influences of a overloaded display against the possibility of omitting crucial details.

An intelligent system should also handle the pilot's inputs in a flexible way to provide some kind of DWIM (do what I mean). So, e.g. ambiguities arising from cursor positions that are not uniquely correlated to display points may be resolved using some knowledge about the situation.

An expert system for mode and display control may be a dedicated system (pilot's associate) or part of the avionics system in concern (radar, ESM, navigation, control).

2.1.2. Decision Support

An Expert system can give direct decision support for the pilot (and eventually the WSO). This can be done by proposing the best decision to the pilot or by showing those facts that are needed to make a decision. An expert system can combine a lot of facts, gathered over the time of the whole mission or requested from other sources. It may select the best decision or condense the facts to an amount that can be displayed in an ergonomic way in a combat situation.

Decision support comprises warning and notifying the pilot on situation changes. This becomes most important when the attention is distracted in critical situations or in the course of routine surveillance tasks (e.g. in a Combat Air Patrol). Such decision supporting systems need to be aware of the situation in order to assess the importance of incoming observations.

2.1.3. Situation Dependent Briefing

Crew briefing is essential for the success of the whole mission of a fighter A/C, but is limited by human ability to memorize facts. Too great an amount of details and facts that are not likely to be needed may confuse the crew and degrade their ability to remember the essential things. An XS may brief the crew only on those facts that are important just now, depending on time, location and tactical situation. ("By the way, .. ")

2.1.4. Monitoring and Surveillance

Maintenance support and testing is mostly based on heuristics since complete testing of all possible constellations and inputs is not possible in a complex system. Build In Test (BIT) for isolated systems may lead to a failure and fault isolation in the system. This may be performed by an expert system and an intelligent BIT may not only detect and locate failures on the fly but also estimate the impact and consequences and provide graceful degradation by changing configurations or calling for adaquate countermeasures. This may be done by an autonomous decision or by giving an adequate advice to the pilot.

The surveillance function also comprises the tasks of fuel and coolant status control. The controlling system has to decide from the overall situation whether a shortage may occur or is critical and whether the shortage is important enough to notify it to the pilot thereby possible distracting his attention from more important observations.

Medical surveillance is another application that may neccessitate expert systems application. Medical systems have been among the first expert systems. Since man is not fairly rapresented by technical or mathematical models, heuristics play an imortant role in medicine. Medical surveillance in avionics systems may comprise long term surveillance in order to detect when the crew is getting unconcious, becoming overworked or is hast lost the orientation. It will also comprise short term manoever surveillance e.g. in high-g situations where the consequences may affect the mission.

2.2. Autonomous Decisions for Radar Components

We want to discuss nose radar oriented tasks as an example for the many sensors that are on board of a modern aircraft (navigation, infra red sensors, air data, altimeter). Such an XS may either be a separate dedicated system or be part of the radar system (e.g. the radar data processor). A possible configuration is given in figure 4.

Many tasks occur in parallel also in electronic warfare components. ESM (Electronic Support Measures) tasks are similiar to those of radar and other sensors and will be omitted here. ECM (Electronic Counter Measures) emission (e.g. jamming) especially needs clever and tricky reactions and will be treated in brief in the mission oriented section about self-protection. The cooperation of radar and ESM/ECM components is a mission oriented task that concerns the whole weapon system and will hence also be treated in the subsequent section.

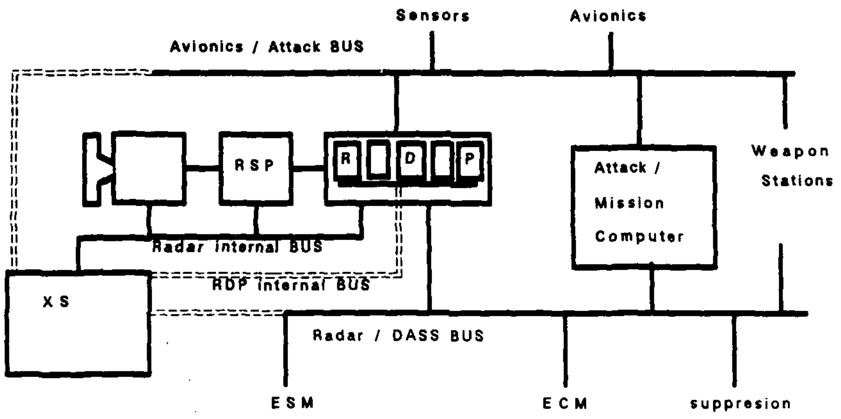


Figure 4.: Possible location for a radar oriented XS. The connections to the avionics, attack and defensive aids subsystem busses would provide the possibility to handle also mission orientd tasks.

2.2.1 Mode Selection

Modern sensors such as airborne radars provide a lot of modes, sub-modes and parameter settings that can be selected on flight to achieve maximum performance and mission success. The optimal mode depends on external data, pilots intention and mission aim and parameter selection requires experience and knowledge about the system's functions. Expert systems may support the pilot or weapons system operator (navigator) in their task of selecting the most appropriate mode and the best parameters for the radar system. The task is to select the radar mode, radar parameters such as beam splitting beam shape and width scan pattern, width, speed, pulse width and pulse rate, RF and frequency agility, and the power on and off targets and the false alarm rates and thresholds. The selection is based on radar information such as track histories, actual signal noise and false alarm rates, on avionics information such as ownship motion and manoever, weapon mode (gun, missiles, bomb guidance), and on ECM activities. The overall tactical situation of own and hostile SAM and troop deployment and the airborne enemy and own forces deployment, the position of the wingman, the type of the mission and the weapon and fuel load will also influence the optimal radar mode.

A mode selecting system needs knowledge about the interpretation of track history, noise, jam and tactical situation. It must be able to assess the detection probabilities as a function of radar mode, beam splitting, shape and width, scan pattern, width,and speed, pulse width and pulse rate. The system must weigh up the neccessity of accurate measurement against that of extensive information gathering and the neccessity of accurate tracking hostile targets against the disadvantage of being detected.

2.2.2. Intelligent Tracking

A system for filter control may assess track, plot and target noise and systematic deviations (error, manoevers, ECM) to supervise and control filter gains, adapt them to the external noise, and eventually reset Kalman filters. Plausibility control may detect manoevers, ECCM, and multi-target situations. Clever track file handling comprises criteria for the initiation and deletion of tracks and a priorization logic. The insertion of tracks from other sensors as well as intelligent decision for plot-track correlation in many-to-many situations need the knowledge about manoeverability, tactics, own sensors's accuracy, and track-history interpretation. A ground based expert system controlling Kalman filters for tracking has been tested successfully.

2.2.3. Priorization

The priorization and grouping of targets (a/c, ground sites, tanks) according to their importance as a threat and as a potential target is primarily based on the Inverse Time To Go, i.e. the quotient of range-rate and range. Other criteria may be target features such as manoevers, flight direction, friend /foe classification and the target's weapon mode as seen from the radar warning reciever. Also the available weapon and fuel load and the mission type of

the ownship as well as the overall tactical situation determined by the deployment of own and hostile aircraft and SAM sites have to be taken into consideration.
An example for a priorization that provides good situation avareness and optimal opportunities for weapon deployment is given in figure 5.

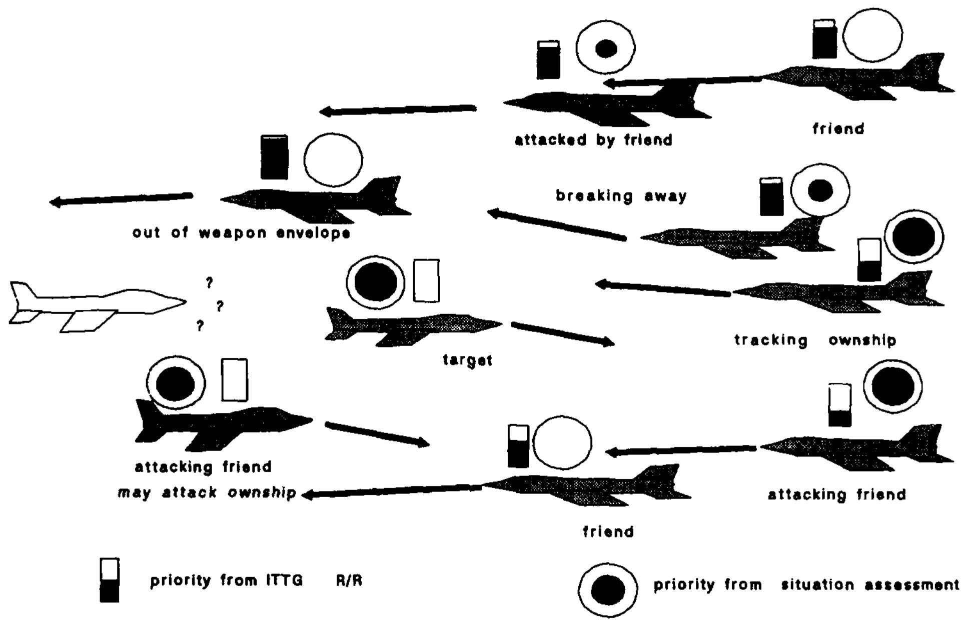


Figure 5.: Intelligent priorization involves situation assessment.

2.2.4. ECCM

To counter hostile countermeasures needs cleverness in order to cope with the game theoretic situation that the enemy also reacts to your reactions. He may foresee your reaction or may even force a special countermeasure reaction that has advantages for himself. The reaction of a countermeasure or counter-countermeasure system should take into account all relevant past events, especially on the hostile reactions to own measures, as well as the own and hostile intentions.

Plausibility control of incoming data and results and the fusion of several independent sources is a good form of countering countermeasures and it also provides the system with the possibility to degrade gracefully by leaving away suspicious and possibly misleading data.

2.2.5. Multi Sensor Fusion

Multi sensor fusion describes the task to combine incoming data from several sources such as ESM, ECCM, FLIR, IFF, etc. The idea is to achieve synergetic enhancement of the heterogeneous sources of information, but problems arise since the data may be inaccurate, inexact, vague, incomparable (e.g. different physical parameters), or even contradictory. Multi sensor fusion may be used for tracking, identification and navigation and it is located somewhere between sensor and mission oriented tasks since it may involve a lot of sensors.

2.2.6. Identification

Identification is the task of correlating an object to one of several classes based on observations. It seems a simple task that can be performed by any intelligent creature but identification is not easy to implement in a data processing system since data may be incomplete, inaccurate or contradictory. Non Cooperative Identification (Target Recognition) describes the task to identify an object (e.g. an aircraft) based on the informations that are available from the aircraft's sensors. Its combination with cooperative identification (secondary radar) makes the decision more secure but much more complicated.

Identification and classification is allways based on heuristics, even if there is some mathematical theory used to transform the heuristics to numerical algorithms. The basis for the analytical (nearest neighbour, linear or polynomial classifiers, cutting planes), stochastc (Bayes) or optimization (sequential decisions) methods is allways the heuristic that assumes that this approach gives "best" classification. The selection os the method, of error and convergence factors, optimality criteria and template size is only based on experience. Heuristic programming seems to be the best way to accomplish identification tasks. While analytical methods are rather inflexible assuming a fixed set of incoming data, the stochastic optimization approaches are very complex, hence time-consuming. Classification algorithms need a lot of data in the learning phase since they lack the ability to abstract and to generalize. Since expert systems involve concepts and notations used by men, they provide a more flexible access to the design of identification systems. This will be discussed further in the chapter on the DAVID identification system.

2.3. Mission Oriented Decisions in the Weapons System

Mission oriented tasks involving optimal behaviour in game theoretic situations, weapon deployment, and automatic pilot may lie in the far future. Nevertheless there are a lot of applications of heuristic programming that seem to be possible in the near future and we want to scetch some of them.

2.3.1. Manoever Oriented Tasks

Manoever oriented tasks comprise short-term decisions in manoevering such as high-g-avoidance, automatical steering, or variable sweep wing control for manoevers. On longer terms we have smart terrain following and terrain contour masking, combat manoevers in Air-to-Air and Air-to-Ground (e.g. countering SAM sites), tactical manoevers for threat avoidance, manoevers for optimal sensor accuracy commanded by radar components (ELS, passive tracking) or manoevers for weapon deployment.

2.3.2. Electronic Warfare

Some of the load of coordinating several ESM and ECM systems and weapons can be taken off from the pilot. Optimal balancing of RF self protection with self protect weapon delivery involving power management, HOJ (home on jam) avoidance, frequency management and multi a/c emission control could be performed by an expert system.

2.4. Ground Support

Ground support will be among the first aplication areas for avionics expert systems. Since many tasks such as maintenance or configuration are not specific for this application area, we will omit them at all. We will sketch in brief those applications that are typical to avionics.

2.4.1. Mission Preparation and Briefing

Expert systems for mission planning comprise Smart Terrain Following and Terrain Contour Masking planning, and fighter deployment and schedule, e.g. to achieve best radar coverage for a Combat Air Patrol. Data fusion to the essential informations is needed for the crew briefing as well as for the briefing the on-mission-briefing-XS discussed above and for the debriefing and post-mission-analysis.

2.4.2. Ground Programming Support

Ground programming support comprises the generation, test and modification of emitter libraries for ESM, ECM, and ECCM; of parameter libraries for sensors, electronic warfare and weapon components; and of identification lists, templates, and classification parameters for identification tasks. Problems arising for example from ambiguities in emitter libraries are at the moment handled by human experts but there exist no algorithmic approach. This indicates that these problems are a suitable application area for expert system altough today's expert system shells and tools are not able to handle the complex notations needed to handle overlapping in a multidimensional parameter space.

Programming the knowledge base of an airborne expert systems can be seen as part of ground programming support. On the first look, it seems much more difficult to program an XS than to generate a simple parameter list such as the emitter or identification libraries mentioned above. But we have to consider that there is much more ability to dissolve ambiguities in a knowledge based system than in a primitive system working on a single parameter list and that libraries used e.g. for search regime determination and emitter identification for RWRs or ARMs have a more complicated structure and contain more "knowledge" (implicitly) than many of the expert systems known from the AI-literature. From this it seems plausible that a system to write a rule-base for an XS is not more difficult to develop than a systems that automatically generates and compiles a parameter list with the same performance.

3. Problems in Applying Avionics Expert Systems

The main obstacles hindering expert systems from a widely accepted use in avionics systems are their lack of real-time capabilitiy and the problem that expert systems seem to be non-deterministic and non-verifyable.

3.1. Real Time Expert System

The following aspects make up the "real-time" capabilities of a computer system, hence determine its useability in airborne applications:
- fast (average response time)
- guaranteed response time
- guaranteed handling of all incoming information

Restrictions regarding the average response time may be overcome by increased HW speed and/or by massive parallelism which is easier implemented in knowledge based systems then in conventional software systems. Airborne XS processors may be MIL versions of commercial systems such as the MIL version of the TI explorer, SUN 3/50 (ARX 3/50), IBM PC/AT (TAC-PC) or the DEC MICROVAX II (MIL VAX II); ruggedized commercial systems; or dedicated systems based on transputer arrays or microprocessors (Conner,1987).

Guaranteed response time and the guaranteed handling of all incoming information is a problem of any higher order language, it may be overcome by AI-language with the capability of handling interrupts and timers. An integration of interrupt handling into a AI systems would be a task that only concerns the operating system. One should keep in mind that with such time varying fact bases, PROgramming in LOGic will no longer be pure logic and the behaviour of such systems must be testet very thoroughly.

Guaranteed response time might be integrated into the language, for example by a way to tell PROLOG not to take more than a given amount of time for a special task. For example, a Real-Time-PROLOG expression like
R :- ((P1 (t1), P2 (t2))(t3) ; P3(t4))(t5).
might say that R is true if P1 and P2 can be proven in time t3, while giving P1 not more than t1 and P2 not more than t2 time, or if P3 can be proven in a time of t4 while taking no more time than t5 for considering this statement for R. Note that the time given to prove a statement (e.g. P2) is given by the minimum of the constraint stated explicitly (i.e. t2) and the time left from higher order constraints (i.e. t3 - t1' where t1' is the time needed to prove P1). In situations where nesting, iteration or recursion may occur, precautions must be taken so that searching time is not spent on the lowest level or the first goals while leaving no time for the recursion to come back. This may be achieved by making the maximum admissible time an explicit function of the time left over for the calling predicate. In our example, the formula could then read R(t):- ... where t5 = t, and t1,..t4 are functions of t, e.g. t4 = t/2, t3 = 2t/3, t1 = t2 = t/2.

3.2. Proving Expert Systems

It is often said that AI-systems are not deterministic since the knowledge base does not determine the behaviour. The same may be said about classical real time software in connection with the runtime system and it is clear that an expert system can only be verified in connection with its inference engine. Although XS are not easy to verify, they may be more easily validated since they provide a more direct connection between the problem solving knowledge (specification) and the implementation. The problem of testing an XS, i.e. controlling whether it provides the correct answer, is critical in situation where no one knows which answer is the right one. The problem of responsibility for XS decisions in critical situations (including weapon deployment) is as critical as the responsibility for classical program's decisions and aircraft crew decisions.

Standards for software development such as DOD-STD 2167 or AQAP 13 may be transferred to the knowledge engineering and XS development process. Maybe a validated and standardized XS-shell or toolbox will help to make software development for complex systems safer and cheaper and will significantly decrese modification and maintenance costs.

4. The DAVID Expert System

To demonstrate the applicability of the expert system technology to avionics tasks, a demonstration system for the visual identification of combat aircraft (Demonstrator for Aircraft Visual IDentification) was developed. The task was to identify about 100 aircraft types by means of the textual description of a single picture. After a time of about 6 month of knowledge acquisition and structuring, system design and prototyping, a system that could identify 25 fighters and attack aircraft using about 100 rules was implemented. The number of aircraft in the final DAVID system will be bounded only by the user supplied data.

4.1. Task and System Design

DAVID demonstrates the applicability of reasoning under uncertainty, using fuzzy logics and uncertainty modelling. Its tasks parallel to some amount the tasks that have to be accomplished by a NCI-module as described above. Visual Identification was chosen since the system was intended as a demonstration system and dedicated to men whos^ most important sensor is the eye, not to radar. We also wanted to have a proven expert and to see in case cf a wrong decision whether the reason for a failure was inherent to the system or caused by data that were either wrong or insufficient for an identification.

This controllability was eased by a textual interface using a menue of possible answers (in forthcoming phases supported by graphical output), so no tasks of pattern recognition, image recognition or picture analysis had to be performed by the system. This was also selected sice a menue-based textual interface allows to replay session dialogues and hence to control the systems decisions and to assess its performance by comparing them to the ability of a human observer working with the same data.

The target classes of the identification process were the types of the aircraft, so we had sharp (non-fuzzy) man-made classes, while other classes and notations used in the process of identification were fuzzy. Especially in combination with the human observer there was the problem that in most attributes we had a continuous transition between several values of the attributes rather than several well distinguished values.

4.2. Knowledge and System Description

4.2.1. Knowledge Acquisition and Representation

For the knowledge acquisition, structuring and representation we adapted a phased approach proposed by Harmon and King (1986).

In the orientation and problem definition phases, requirements were collected and after identifying an appropriate expert and the first discussions we derived a rough concept of the domain and the problem space and a project plan for the next development phases. In the analysis phase the collection, analysis, and structuring of the concepts and terms of the expert and of the user led to a overall system design.

From the beginning, the development was intended to have two major cycles: a small prototype using a PC-tool and a stable system implemented on a DEC VAX. Both systems had to run through the phases of design, implementation and the loop of testing and improvement.

The identification knowledge and problem solving methods were gathered from various discussions with the expert. The formal expert interviews in the process of knowledge aquisition and -analysis could be kept at a small amout since the expert knowledge was important mainly for the procedures and approaches while most of the fact knowledge on the different aircrafts could be taken from the literature. Nevertheless, the system was discussed with the experts in all stages in order to stay on the relevant attributes and with the best problem solving behaviour. Any discussion round in the design, development and test phase necessitated some rewriting of the rules.

One of the best tools was the graphic representation of all relevant objects and relations (semantic net) which gave a good basis for the discussion with the expert.

4.2.2. The Representation of the Fact Knowledge

The experts knowledge contains much more information and structure than a simple list of attributes or a set of relations or the simple "If air-intake is large and the aircraft has variable wing and twin fin it's a F-14 Tomcat" explanations given by the expert. Expert identification is to a great amount based on a model of the aircraft and on technical knowledge. The connections between manoevrability, fins, power, and air-intakes are present in this knowledge as well as the typical properies of variable geometry wings and a model of the aircraft that allows him to judge whether twin fins may be hidden by the fuselage on that special picture.

Part of the semantic net for aircraft identification compiled in the course of knowledge acquisition is given in figure 6.

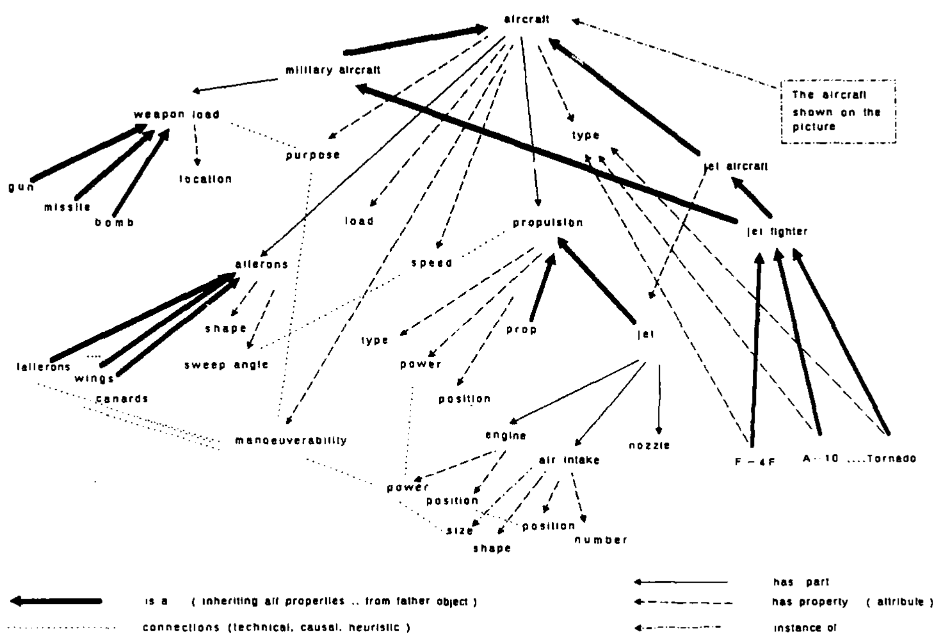


Figure 6.: Some of the notations and connections relevant for aircraft identification.

The frame-like hierarchical object structure of the aircraft identification given in figure 7 is much closer to the implementation than the semantic net.

4.2.3. Problems Identified

In the course of the knowledge aquisition and -analysis some problems in representing the relevant knowledge occured which we want to scetch in brief.

It is very important to distiguish between hidden, concealed, absent, not identifiable, and not describable attributes. For example the fact that the air-intake is hidden by the wings brings more information than e.g. the fact that the air-intake is round, but the first information is more difficult to represent since it strongly depends on the observers aspect.

The difference between fixed and variable geometry wings can be seen by an observer. Here, a helping facility and metarules are important. Fixed geometry is easy to judge in case of a delta wing, but in the case of swept wings difficulties arise. Depending on the geometry at the time of observation, variable geometry can be judged from sharp bends, a broad shoulder in combination with swept or straight wings, or a delta like wing shape formed by aileron and tailplane.

The notations, concepts, and terms differed widely between the experts and the users and among several users. In our first system with small menue, almost any of the possible answers for the F-4F's air intake form and wing shape occured. This was overcome by adding special values in the menue, separating user's and expert's notations, and implicitly allowing fuzzyness (round-oval-...). The notations of the expert and the user were connected via dedicated rules that gave plausibilities e.g. for a swept wing when the user selected a straight wing or vice versa.

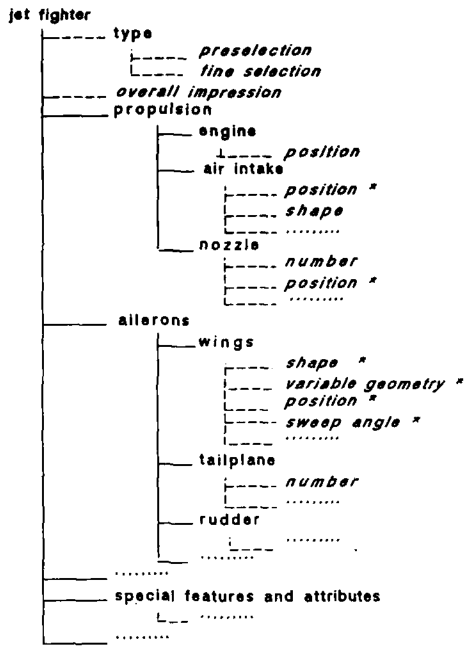


Figure 7.: Hierarchical object structure used for DAVID. For the attributes with a *, the user input notation and system notation are separated.

4.2.4. Problem Solving Knowledge

If, after the consultation, one asks an expert how he got his results, the answer may have the form "if the aircraft has a V-shaped twin fin and if air-intake is round and underfuselage then it's a F-18 Hornet". A set of such rules will never build up an adequate identification system. On the one side it lacks knowledge as discussed above an one would need 10 to 20 rules per aircraft. On the other side, the dialogue between the user and the expert (-system) must first concentrate on rather general properties. The user will not accept 30 questions on special attributes that must be answered mostly with "no" or "not applicable". The Expert system must have some structure and control to run an intelligent dialogue.

Some of the identified techniques are listed below:

a) Heuristic of successive refinement

The expert looks at the parts of the aircraft to get an impression ad a pre-selection of the type. Most times, the standard parts with their properties are asked first, for example according to the following list:

- wing assembly
- tail unit
- propulsion, engine, air intake and jet
- airframe, fuselage

after this, special parts such as canards, antennas and fuel, and electronics pods and weapon load are asked for. Another sequence is given by the "WETFUR" Wings - Engine - Tail unit - Fuselage - Undercarriage - Radiator mnemonics (Wood,1982).

b) Exclusion and confirmation by means of special attributes.

The attributes asked for according to the a-priori knowledge (when the user describes a fighter aircraft he has seen, in Germany in most cases it may be a F-4F) and on the a-posteriori information from the questions before.

c) "what catches the eye"

The expert lets the user describe some properties of the aircraft. Sometimes this is initiated by the user who starts with a question for an aircraft with some special property. The implementation of this technique is rather difficult since a free textual input from the user requires some association logic, while a text or graphics menue would propose much more than the 7 items a human observer can deal with.

d) staged and switching approach

Experts have the ability to combine several problem solving paradigms to get optimal results. In reality, the identification process combined the three methods listed above by switching to the most appropriate one.

4.2.5. Concepts of Uncertainty

The expert does not explicitly think in uncertainties. Hence, it is quite difficult to determine which uncertainty representation is the most appropriate one. The crucial test for this must be the behaviour of the resulting system.

Simple probability factors (e.g. with additive combination) and the certainty factors provided by TWAICE provided an easily handeled method but sometimes it lacked flexibility. In many cases, a linear scale of plausibility does not adequately describe all the knowledge on the chances whether the fact is true or false. Moreover, different kinds of rules need a different way to combine the certainties of their premises and the rule to get the conclusion's certainty factor. The same problem arises when facts are derived from several rules and the cobmined certainty factor should depend on some degree of independency of the rules. Finally, the certainty factors do not allow non-monotonic reasoning. If an intermediate fact in some chain of conclusions has a plausibility of, say .5, no evidence can raise the plausibility to more than .5 while on the other side, no independent counter argument can get the plausibility below this value.

Bayesian representation of the dynamic knowledge (consultation inputs as opposed to the static knowledge in the knowledge base) seems to be concise and flexible enough. The Bayesian approach would give a probability to any type. The problem is to determine adequate a-priori probabilities, to gather and represent the answering probabilities which may depend on the aspect, and to do the necessary calculations of the a-posteriori probabilities. Sequential Bayes decisions (in the form of a Markovian Decision Process) with the objective of minimizing costs occured by misclassification and identification effort was considered an alternative for the system implementation. This approach would take the a-priori probability and would determine the optimal question by evaluating the expected cost from the a-posteriori probabilities arising from the possible answers. Altough the calculation effort is too high, Bayesian optimization provided a very good model for the explanation of the experts behaviour (e.g. the methods switching discussed above).

Fuzzyness turned out to be a flexible approach and was rather usefull in the first stages of identification. Fuzzy sets were used as a representation of the set of possible aircrafts after the first identification phase. The connection between user's and expert's notation was based on the idea of linguistic variables.

Using a 2-dimensional space of evidence for any possible aircraft type in order to combine the evidence for and against that type may be seen as a rough view to bayes estimate, as a special kind of fuzzy set, or as the simultaneous Shafer-Dempster belief function for "yes" and "no". Whether the evidences should be combined like probabilities or belief (i.e. mainly multiplicative) or like (fuzzy) logik (using minimum/maximum) depends on the kind of rule applied. We considered it the most adequate method under the given time and space constraints, but to process it some kind of non-monotonic reasoning is neccessary and this was not provided by the shells used.

4.2.6. System Implementation Approaches

The basic approaches for a rulebased system were:

- one large rule per a/c
- one large rule per object-attribute-value
- a set of processing rules (metarules) plus a database containing the facts

To use one rule per a/c reflected the "IF that AND that AND ...THEN it's this type" explanations. There are two principal approaches: either to use only such attributes that will discriminate the aircrafts very effective or to use a lot of attributes that are the same for any aircraft. While the second apraoch produces lengthy rules but runs faster, the first one makes the consultation more lengthy asking a lot of questions on very special attributes. If there is no possibility to weigh the attributes and to use and/or combinations in the premises, about 10 to 20 rules may be neccessary per aircraft type. A superrule used for this approach reflected something like a frame or form that contained all possible entries (objects and attributes). It was very helpful in the development process where writing a rule was reduced to cutting down a superrule to the needed attributes and filling in the correct values.

To use one rule per object-attribute-value gives some fuzzy-set or Bayes-like approach. The knowledge on one aircraft is widely spread and the course of the consultation is hard to control. If no precaution is taken, this leads very quickly to senseless questions and to overloaded menues. The superrule used for this approach reflected something like a list of all possible aircraft types.

To use a database or list representation of all the facts combined with meta-rules for the consultation control and information weighting seems to provide a good solution, but it's rather difficult to implement. The database structure needs to be very flexible. To control the questions in a rather general manner seems not possible since the structure of the knowledge is flexible and depends on the dynamic knowledge itself. This approach may end in an intelligent search through a tree-like structure and may represent an "expert recognition handbook user" instead of an identification expert for aircrafts. Nevertheless, this approach may be used to implement an already developed system. It also may be used to produce a widely useable identification shell which will be easily transferred e.g. to mushroom identification but will contain no facts or problem solving knowledge of the special area.

4.3. Implementations

The system to be implemented should be dialog-oriented, and must comprise concepts of uncertainty. The numbers of types a system could handle will be easily increased by a factor of 2 to 10. It is given only to have a scale for the numbers of rules and rule-classes in the system.

4.3.1. A Flat Rulebased System

This system was intended as a first prototype to test the feasibility and acceptance of DAVID on a low cost level (less than one week implementation effort). It was build up using a typical PC-shell (PICO) which did not allow to structure object, handle incertainties or use metarules. Hence, lot of the knowledge structuring done before could not be carried over. The XS was implemented as a flat system (any rule was directly connecting input data with the result) using rules that contained all relevant attributes with the possible values for one aircraft.

At the first approach, we used 3 rules per aircraft representing the front, top and side view used by aircraft recognition handbooks. In the stage where we stopped working on it, the system could handle 22 A/C using 99 rules. About 4 rules per A/C were used, covering combined top, bottom, front, back, and side aspects and special visibility conditions. The set of attributes asked in the rules was a compromise between the need to have a lot of information to achieve a sure identification and the problem that a single hidden object or an inaccurately described attribute would let the rule fail. Rules with a smaller set of attributes were given a lower certainty which implied that they were not invoked when there was a better rule.

Any rule had to be seen in the context of all the other rules. Hence, adding a new aircraft to the rulebase, adding a rule for an existing aircraft, or changing a rule yet contained in the knowledge base might result in rewriting a lot of rules. Much care had to be taken to those rules that tried to identify aircrafts with a small amount of items or from an uncharacteristic point of view.

To give an example, the system contained 5 rules for the F-4 Phantom, each used up to 10 attributes. All the rules used general features such as the engine position in the fuselage and the side position of the two air intakes. This was a general design principle in the development of the rule base which helped to avoid a large amount of very special questions. Two rules were mainly dedicated for the front view, mainly on air intake and the position and form of the wings. Two rules were mainly dedicated for the top or bottom view, mainly on air intake and the wing shape. These rules were duplicated in order to allow different answers for the air intake form since PICO did not allow to formulate an OR in a rule and the air intake form was not unique (rounded rectangle or oval, the later answer had a lower certai..ty factor). One rule was mainly dedicated for the side view and included the jet position. For aircrafts with a rather similar counterpart, more and larger rules were needed. So for example 5 rules with up to 15 attributes were used for the MIG23/MIG27 Flogger to separate from the rather similar F-111 and MRCA Tornado.

The prototype was presented to our expert partner. The system provided a facility to store consultation data which built up a case library for intensive testing of the XS. Several experts and a lot of interested people used the system which led to an ongoing improvement of the rule base.It soon became apparent that a primitive shell would not suffice for our needs. In order to accomplish a fully satisfactory result, up to 10 or 20 rules per aircraft might be necessary. Nevertheless, working with the prototype has brought a lot of insight, fact and structural knowledge and has opened our eyes for the problems connected with the identification process.

4.3.2. Staged Identification

With Nixdorf's XS-shell TWAICE, version 2.5, a system was implemented that represented the experts knowledge and problem solving procedures in a much better way. The system was characterized by a staged identification approach, the explicit transition between user notation and true world (or experts) facts, and by the handling of uncertainty.

With its present knowledge, the system can identify 22 A/C. This number is not determined by system constraints but only by the state knowledge base. These aircraft are all fighter a/c and neither delta-winged nor prop driven a/c are contained, so an enlargement of the knowledge base to other types would not significantly enhance the identification problems.

Rules represent the facts about the aircraft types, the problem solving knowledge is implicitly contained in the sequence of the rules, of the attributes in the rules, and in the stages of identification.

The 1st stage builds up a fuzzy set of possible aircraft types as an intersection of the fuzzy sets build up from common attributes. At the moment, 7 fixed attributes are used. Since the questions are context sensitive (e.g. if the air-intake is not visible, the form of air-intake is not asked), between 5 and 7 questions will be asked.

On the 2nd stage the investigation concentrates on those aircraft that remained in the fuzzy set of possible ones. These are tried to exclude by means of common attributes. For any aircraft, 4 to 6 attributes are used in context sensitive questions. The attributes used for this stage have been chosen to be mostly the same in order to provide a short overall response time.

The system contained in this stage 90 rules, among these were

22 rules for combining the notations of the user and the expert, for fuzzy terms, and for the handling of unknowm or hidden attributes (e.g. "if air-intake is invisible then form of air-intake is unknown"),

36 rules for the first stage fuzzy set determination (one rule for any value of any attribute, any rule has as much conclusions as there are possible A/C for this value of the attribute) including one rule for the fuzzy set intersection ,

22 rules for the 2nd stage exclusion by means of common attributes (One rule for any A/C, any rule has as much propositions as there are attributes to be looked at),

2 rules for output control (printing rules),

8 rules for general control such as automatic instantiation of objects

The taxonomy contained 7 objects and in total 43 attributes with about 250 values among which the immediate a/c properties and the user inputs each used about 10 attributes and 50 values (cf figure 7.). The aircraft types in the two different identification stages and in the 8 preselection fuzzy sets formally added 23 types each time.

Two further stages (called 3rd and 4th stage) are just in implementation and will each involve about 30 special attributes for a more concise (3rd) or a definite identification (4th stage), each using one rule per aircraft. Three rules controll the identification procedure. Depending on the number of aircraft left after the first and second identification stage, the system continues either with the routine questions or asks for the special attributes connected with the remaining aircrafts. This parallels the expert who aks more specific questions if there are less possible types and anks some test questions if only one possible type is left.

4.3.3. Integrated Approach

An implementation that is closer to the expert's notations is planned. It will be implemented using the new version 3.0. of TWAICE which provides some features of object oriented programming. Alternative, a workstation based shell such as NEXPERT or EPITOOL or a hybrid tool such as LOOPS or KEE may be used. The system will combine facts, objects, and problem solving methods.

In this system, the rules represent the knowledge about problem solving. Facts about A/C are contained in the hierarchical object structure and in the object's attributes. The object hierarchy is such as in the semantic net. The values of the attributes used in the identification process will be kept in a database. This allows to do minor changes or addition of a type which fits into the present sceme without changing the expert systems rule base. The rule base or taxonomy of the system has only to be changed if a new expert behaviour is detected or a new aircraft is added which is structurally different fron that known before. The data base must be flexible enough to allow structural modifications without necessitating a reload of the data.

The processing of the rules can be controlled by metarules and supervised by a rule browser. This provides the flexibility to change the problem solving strategy from general to special questions and vice versa depending on the information available at the moment.

The new system will have a graphical interface consisting of a combination of canned pictures and elementary graphics. This will allow to produce menues e.g. for the possible shapes of a wing or to give the user a tool for the input of span, angle of sweep, and locations relative to the fuselage by using small pictures and a display oriented device such as a mouse.

4.4. Lessons Learned from DAVID

4.4.1. Lessons on Expert Systems

Expert systems provide a powerful tool to solve decision problems for which no algorithmic solution exists. By rapid prototyping (modelling) and working with the knowledge, a much deeper insight is gained. Even when time or space constraints or the need to do a lot of conventional dataprocessing (number crunching, signal processing, numerical algorithms) force to use a high order language or even assembly language, the expert system may serve as a basis for a conventional program.

Expert systems are easier to assess than conventional decision methods. While it takes a lot of experience to see which parameter is responsible for some special behaviour in a Kalman filter, polynomial classifier, or linear programm and what must be changed in case of malfunctions, in a knowledge based system the connection between the behaviour and the rules and facts is rather clear. Even if the behaviour of the system cannot be easily seen from the knowledge base, unexpected reactions can mostly be traced down to some pices of knowledge.

This also faces the knowledge engineer with the problem that the user wants to see or even write the rules. This in critical in those cases where a lot of structuring had to be done in order to implement the system. It brings also the danger that the expert begins to think in rules or changes his behaviour.

The applicability of today's shells is bounded by their slow reaction time and knowledge representation mechanisms that are sometimes not powerful and flexible enough. Of course there is a tradeoff between the power of a tool and the teaching necessary to use it, but at the moment, all tools are far from the optimum. Also some additional features are neccessary. so, e.g. non-monotonic reasoning is necessary ("if air-intake is not visible then a/c is F-111 rather than MRCA Tornado") and a shell or tool should provide an easy approach to outside data and programs and a flexible way to combine uncertainties.

4.4.2. Lessons for an Identification System

We think that the staged identification combining standard and requested data provides a good approach to identification tasks whose structure may be carried over to radar NCI. In this application, the standard data (as used for the first fuzzy set and the following step) may be all routine data from the sensors such as radar cross section and echo modulations from the radar and the ESM classification of the target's nose radar that are handed to the expert system in every identification cycle. If neccessary, the radar may request more data (e.g. track history), further measurement (e.g. at another frequency) or further evaluation of the measured data (e.g. peaks of some autocorrelation spectrum or the jitter of the nose radar's pulse repetition intervalls). We think that DAVID may be transformed to serve as a basis for an Ada program for non cooperative identification.

5. Conclusion

For tasks of the complexity needed to provide usefull (not toy) avionics expert systems, a shell or toolbox doesn't exists and will - in the authors opinion - hardly will exist in the next few years. It may be worthwile to build up a dedicated shell, say for display handling, multi sensor fusion, mode selection, or gracefull degradation tasks in order to use it in several applications but it is unlikely that a commercial shell or a stripped avionics XS will satisfy for such a task. It may be a failure to think that good XS will ever be created by an "intelligent tool" transferring the experts unstructured peaces of information into a bag of rules.

A good software engineer plus an excellent support environment and concise ideas and concepts will perform better than an inexperienced user supported by a highly sophisticated XS-tool.

Expert systems are not the only way to achieve these goals, the XS technology should be combined with other AI techniques such as pattern and image recognition or automatic learning, with conventional decision techniques such as mathematical programming, game and risk theory and with software engineering principles.

Nevertheles. XS methods are needed to solve tasks where expert knowledge, heuristics, concepts, and knowledge representation and processing are important to solve a problem and for this approach XS-tools are needed to support the software- and knowledge-engineer in his work of knowledge-acquisition, modelling, structuring and rapid-prototyping. The amount of work necessary to come to a perfect systems would be much greater without the support of good tools and paradigms. For airborne applications in the near future, XS may primarily be used as a tool to develop prototypes, gaining deeper insight into the problem, and producing a well established development baseline that will either be implemented in conventional software or may be running in some stripped form in a real-time environment.

6. Literature

McTigue,T.V. (1982) F/A-18 Software development - A Case Study. In: AGARD AVP Preprint No. 330 Software for Avionics pp 39-1 - 39-15.

Buchanan,B.G., Shortliffe,E.H.(1984): Rulebased Expert Systems. Addison-Wesley, Reading, Mass.

Conner, M.S.: Low-cost, rugged commercial computers fit military needs, EDN, December 24, 1987.

Harmon,P., King,D.(1986): Expertensysteme in der Praxis. Oldenbourg, München. (orig: Expert Systems - Artificial Intelligence in Business, 1982, Wiley)

Hayes-Roth,F., Waterman,D.A., Lenat,D. (1983): Building Expert Systems. Addison-Wesley, Reading, Mass.

Jackson.R.(1986): Flying Modern Jetfighters. Patrick Stephens, Wellingborcugh.

Newell,A. (1969) Heuristic Programming.In: Aronofski (ed.) Progress in Operations Research ,Wiley, N.Y., pp 363 - 413.

Wood, D. (1982) Jane's World Aircraft Recognition Handbook. Jane's Publishing Company , London.

DISCUSSION

G.Hunt, UK

An identification system can either operate on a single set of information which is "frozen" or on successive sets which are refreshed. In the latter case, these sets may correspond to more detailed images as the target range is reduced. To which of these does the system described by the author correspond?

Author's Reply

The set of information in the course of one consultation is not fixed, but determined actively by DAVID in the course of the consultation. The information gathering is controlled by the procedural knowledge. Although DAVID was intended for the description of one picture and cannot handle temporal knowledge, the insertion of new knowledge from more accurate pictures is possible via the "change answers" feature of the shell TWAICE. For example, you may change the attribute "position" of the object "air inlet" from "unknown" to "not in the nose" and from this to "below the ailerons first third". Each new answer would lead to a rework of the rules by the inference engine, leading to a more accurate identification.

J.G.Brevot, Fr

1. Could you provide more details on the expertise collection in your DAVID expert system: how many experts? how much time did it take? how did you proceeed?
2. What applications do you foresee for expert systems in the near future? What delays are necessary before beginning to have real-time expert systems (XS)?

Author's Reply

1. The expert's knowledge was important mainly for problem solving, e.g., how to proceed and what properties are important. Literature exists for the "facts" data base as well as for the recognition process, e.g., in the German Army AIL recognition Handbooks. Our main expert was formerly with the Air Force. In the first stages, we made case studies and eventually we were able to assess the systems behaviour and therefore didn't have to discuss all types with the expert. The development time was one year.
2. I have sketched some application areas in the papers. Other avionic groups with an increasing difficulty are:

— Groundbased	— development tools
	— support systems
— Airborne	— *decision support*
	— display and moding
	— sensor application
	— mission oriented tasks

XS for real time applications will arise in several steps, each of them bringing the XS-paradigm (heuristic experience-based, knowledge processing systems) and XS-approaches (value based and object-oriented programming, browsers, implementation by specification, etc.) to avionics application:

- XS as a method for gathering in sight and experiences
- XS as a specification for standard language implementations
- compilers for XS to HOL
- real time XS with no learning or explanation facilities
- real time XS with the full power of Artificial Intelligence (AI). Expert systems are now mature enough for the first two steps, the third one is in work, as seen in the conference. Maybe at the next conference we will be one step ahead.

CONTRIBUTION OF EXPERT SYSTEMS TO AVIONICS ADVANCE APPLICATION ON AS30 LASER WEAPON SYSTEM

by

Mr SERGE DUPUY
aerospatiale
2, rue Béranger B.P. 84
92322 - CHATILLON CEDEX FRANCE

ABSTRACT

A brief reminder on Artificial Intelligence and Expert Systems will be made as an introduction. The interest and difficulties relating to this new technique will then be exposed. An example of expert system development for AS30L weapon system will be given to illustrate the points already stated, through this experience. To conclude, the prospects of this technique for avionics will be drawn.

1. INTRODUCTION

The interest in Artificial Intelligence (A.I.) and Expert Systems (E.S's) started in the 80's, when it could be estimated that these new techniques might step from research to industrial application. A.I. is a technique consisting in trying and reproducing on machines, behaviours which should be considered as intelligent, if they were human. E.S's are a branch of A.I.; they are designed to perform the same type of valuation as a human expert in a limited field. In fact, this technique basically uses and concerns knowledge thru data processing machines.

To reproduce on a computer behaviors which can be considered as intelligent, this must schematically be provided with functions corresponding to the main features of intelligence :

- Learning capability.
- Memorization capability.
- Deduction capability.

Failing any one of these faculties, neither man nor machine can be considered as being intelligent, and not even healthy as regards man !

Subjects concerned by A.I. basically include :

- Expert Systems.
- Robotics.
- Decisional aid.
- Voice and signal synthesis and recognition.
- Planning.
- Project management.
- Information retrieval.
- . . .

As the knowledge and experience of human experts fed to a machine may enable non-experts to act like "nearly-experts", avionics diagnosis is concerned by this technique.

Despite the promising feature of this technique, a number of difficulties are identified :

- Technical novelty for which application methods are not yet well run.

- Validation problem for the knowledge bases thus involved (coherence, absence of conflicts, etc.).

- E.S's conviviality and, consequently, intelligence of the man/machine interface. This point relating to the understanding of the natural language is still to be solved to provide all users with easily operable E.S's.

Among its A.I. activities, aerospatiale has developed an E.S. designed for the maintenance of a missile launcher. The basic qualities expected from an E.S's have been checked :

- Memorization.
- Availability.
- Diagnosis explanations.
- Flexibility.
- Portability.
- Conviviality.

This example allowed understanding the general interest of A.I. for its contribution to avionics maintenance, as well as the difficulties it may hold.

2. EXPERT SYSTEMS - GENERAL CONSIDERATIONS

Expert Systems are a sub-set of Artificial Intelligence; as indicated by their name, they perform expertise in a limited domain, an expertise comparable with that of a human expert in the same domain. Hereinafter is a review of main considerations relating to E.S's.

2.1. Basic Conditions to develop an E.S.

To develop an E.S., the following basic conditions must be met:

- One or more experts in the domain are existing.
- The expert(s) are motivated.
- The expertise domain is clearly determined.
- Human expertise in the domain is neither too short (1 hour, for instance), nor too long (1 year, for instance).

2.2. General Description of E.S's

An E.S. is a data processing system whose software usually includes five modules :

- An inference engine.
- A knowledge base.
- A knowledge acquisition module.
- A factual base.
- A user's dialogue module.

Information of the expertise domain are entered in bulk by the expert, via the knowledge acquisition module, in declarative mode instead of imperative mode.

Facts noted by the user for expertise are entered in the factual base via the user's dialogue module.

The inference engine performs deductions relating (inference) the knowledge base elements corresponding to the facts. Thru successive deductions, the inference engine reaches diagnosis. The record of knowledge elements involved during inference, provides diagnosis explanation. The above operating process is quite similar to the human intelligence one.

The new and interesting feature of A.I. actually is the fact that the sequential imperative process experienced until now in conventional programs is broken. Indeed, knowledges are not necessarily entered in a pre-established order and are stacked as received in the knowledge base as the expert transfers the knowledge to the E.S. This evidences the possibilities offered by this novation and the resulting flexibility. It is possible to enrich the system with new knowledges obtained thru reflection or experience.

2.3. Basic Components of E.S's

We shall know review the basic components of the E.S's providing representation of the knowledge, modelization of the reasoning and the data processing frame-work.

2.3.1. Knowledge Representation

Knowledge representation modes

Depending on the subject being processed, knowledge representation in the knowledge base can have various forms :

- Production rules.
- Semantic networks.
- Structured objects (schemes, frames).
- Hybrids.

It should be noted that no universal knowledge representation mode is available, which also means that the "universal" E.S. does not exist.

The representation mode based on production rules is well adapted to diagnosis and was consequently selected for diagnosis E.S. described in further paragraph.

Production rules

Production rules are statements having the following form :

```
IF     <premise 1>
  AND  <premise 2>
  AND  . . . . . .

THEN   <conclusion 1>
  AND  <conclusion 2>
  AND  . . . . . . .
```

Here is a production rule example :

```
IF     <you are chilly>
  AND  <liable to chills>
  AND  <the meteo forecasts a cold spell for tomorrow>

THEN   <you should wrap yourself up tomorrow>
```

This example is deliberately simple and easy to understand by a non expert, so as to better illustrate the form than the substance of production rules.

Each rule forms a knowledge atom; to cover the domain of a mean complex expertise, 200 to 300 rules summarizing the human expert knowledge should be used.

Programming Languages

Knowledge whose representation mode was explained hereabove, is expressed in languages specially designed for A.I. in order to handle concepts (thus knowledge) instead of exclusively digital values. The better known are LISP (LISt Programming) and PROLOG (from French : PROgrammation LOGique). These are the A.I. basic languages.

Furthermore, these languages may be used to set up object oriented languages for knowledge representation as structured objects already mentioned hereabove.

Nevertheless, it can be noted that a number of programs use conventional languages, such as PASCAL and C which, by nature, can easily be applied to a wide range of machines.

2.3.2. Inference engine

For knowledge representation as production rules, an inference engine is usually used. It performs a task equivalent to human reasoning. In fact, this is a program which, for a given knowledge representation mode, is fixed and independent from the expertise domain. When finalized it needs no modifications, whatever the knowledge base evolutions.

To give an idea of the reasoning process provided by the inference engine, let's consider the very simple case of a knowledge base including only the three following rules :

```
R1 : IF  <A> THEN  <B>

R2 : IF  <B> THEN  <C>

R3 : IF  <J> THEN  <K>
```

Rules can be used in two different ways, either in forward-chaining or in backward-chaining. We shall review these two ways hereafter and note the decoupling existing between inference engine and knowledge base, which forms the important technical novation of the A.I. concept.

Case of forward-chaining

Assuming that fact A is ascertained, the inference engine will search in the knowledge base which rule contains A as a premise. By scanning, it finds R1 which contains B as a conclusion. B then becomes a deduced fact (from basic fact A). The inference engine carries on searching to find B as a premise for a rule and finds rule R2 with C as a conclusion. The inference engine carries on searching and finds no rule with C as a premise. It then declares C as a conclusion. The answer to the problem set is that A produces C. The inference engine thus has performed what is equivalent to a deductive reasoning.

Case of backward-chaining

The inference engine operation is inverted with regard to forward-chaining. It goes backward from the conclusion (ascertained fact) to the premise which activates the rules chaining to the conclusion. This premise then is the cause of the ascertained fact, and the diagnosis is obtained.

Considering the same knowledge base as above, assume that C is ascertained, the engine searches for the rule containing C as a conclusion. It finds R2 which hypothetically becomes candidate. The engine then proposes premise B of R2 to the user to confirm that B is an ascertained fact. If yes, the engine carries on tracking to find a rule containing A (premise of R1), fails to find it and stops tracking. It then delivers its diagnosis : the origin of the ascertained fact C is A. The engine has thus made the equivalent of a hypothetico-deductive or inductive reasoning. This operating mode is particularly well suited to diagnosis.

2.3.3. Data processing frame-work

Depending on their importance and nature of their expertise, E.S's may be installed in various types of machines, starting with the PC compatible micros, then minicomputers up to multi-purpose computers and specialized machines for A.I. This represents an extremely wide price range.

It should be noted that, at present, the PC compatible micros represent an enormous field of implementation : this potential market can explain the increasing number of interesting E.S's met on this type of machines.

As indicated previously the use of LISP and PROLOG languages is specially well adapted.

It should also be noted that increasingly performant shells are commercialized. These shells are E.S's with no knowledge entered. Depending on the cases, their utilization may noticeably limit the research and development time and cost for the E.S's to be created.

2.4. Technical Interest

The technical interest of E.S's basically lies in the following qualities :

- **Explicability** : an E.S. gives the reasoning leading to diagnosis, thus contributing to consultant training.
- **Reliability** : an E.S. makes identical reasonings for each case; for a given problem, its deductions are thus constant. The E.S. has an advantage over the human expert : it experiences neither tiredness, nor stress, nor psychological pressure likely to affect its diagnosis.
- **Flexibility** : updating and enriching the knowledge entered in an E.S. is very easy. This leads to development cost cutdown with regard to conventional test systems.
- **Availability** : Utilizing an E.S. allows relieving human experts from repetitive tasks, thus leaving them free for sophisticated expertises.
- **Portability** : As an E.S. or its program are portable, knowledge can be better shared.
- **Memorization** : Using an E.S. permits ensuring knowledge preservation in a Company or Administration. The E.S. has an advantage over the human expert : no risk for memory blanks, even temporary, likely to affect the diagnosis validity.

The above list of qualities, although not exhaustive, can be summarized in one quality : INTELLIGENCE.

Utilization of E.S's concerns increasingly numerous and wide domains : INDUSTRY, FINANCE, LAW, ECONOMY, ADMINISTRATION, MEDICINE, GEOLOGY, CHEMISTRY, ... and obviously, AVIONICS.

2.5. Difficulties

Despite the obvious advantages of A.I. and E.S's, many difficulties of various sorts remain : psychological, technical, methodological. We shall review the most important.

Expert Motivations. As the A.I. basis is the experts' knowledge, their motivation is essential for achieving an E.S. project. Some may fear to see their domain threatened, rejections may then be noted. However, even expertise includes repetitive and non valuating tasks. Creating an E.S. can then be considered as a mean to relieve the human expert from these tasks so that he can increase his knowledge and apprehend increasingly sophisticated expertises. It is an opportunity for him to extend his competences and improve his background.

Knowledge Extraction-Modelization. Knowledge extraction is a technical problem that A.I. specialists involved in a project must solve after having selected : the knowledge representation mode, the reasoning mode and the data processing tools corresponding to the expertise domain concerned. Thru interviews, these specialists will then have to extract the knowledge from the (volunteering) expert then to modelize it and to enter it into the data processing machine. Although not being a major difficulty, these tasks are still now a matter of specialists. Just a minor complication subject : to extract, you have to modelize, and to modelize, you must have an idea of the knowledge, thus you have to extract ! This dilemna is usually solved by developing a simplified prototype of the E.S. to validate solutions and then step to real size system.

Coherence of the Knowledge Base. For an E.S. to have an intelligent behavior, its knowledge base must be homogeneous, without redundancies, contradictions or gaps. Furthermore, during the system life, modifications, amendments, additions, deletions in the knowledge base must not affect this homogeneity. For bulky knowledge bases, checking the coherence still remains a heavy unsolved problem as far as we know.

E.S. Validation. E.S. validation is an important problem, both methodological and technical. Indeed, once an E.S. has been created, the question of diagnosis validity arises; is confidence possible without testing this validity ? How to test it ? Most often, cross-expertises are performed between human experts and E.S. However, this method is relatively incomplete as, due to time and cost reasons, the number of cross-expertises is limited. In some cases for which expertise has a vital importance, validation may be a real problem, not completely solved until now.

Natural Language. In order to give the E.S. conviviality in their interface with the users, permitting to declare them intelligent, it will be necessary that understanding of natural language by a data processing machine be obtained. This is not yet true even in the near future.

Self-Learning. Although research works are conducted in various laboratories, self-learning of an E.S. is still a dream.

3. REMINDER ON AS30 LASER WEAPON SYSTEM

3.1. General

Precision guided weapons greatly improve efficiency and economy of interdiction and close air support missions. In addition, due to the proliferation of accurate short range air defence system, close approach attacks pose unacceptable risks for tactical aircraft, even at very low altitudes. Thus air to-surface weapons must provide:

- pinpoint accuracy;
- firing distances greater than the range of close defence systems;
- aircraft escape manoeuvres just after launch.

Among all guidance modes, laser homing is one of the most attractive answers to the above conditions, and provides:

- outstanding accuracy;
- low cost, laser seekers being the cheapest available;
- versatility, resulting from the possibility of using either airborne or ground based designation;
- firing at night, and even under limited visibility conditions.

Furthermore, laser guidance allows high performance equipment to be used for detection and tracking since these are fitted on board the launching aircraft and not in an expendable weapon.

3.2. Air Force requirements

The mission of Tactical Air forces include close support and interdiction.

Most close support targets are mobile, armoured or unarmoured, camouflaged.

Most interdiction targets are fixed, in well known locations, generally hardened and protected by air defence systems.

Such targets (for instance, buildings, shelters, bridges, dams, air defence radars, depots, etc.) must be destroyed by weapons which have:

- very effective warheads;
- pinpoint accuracy;
- a firing range consistent with maximum detection range;
- the possibility for launch aircraft to manoeuvre just after launch.

To satisfy these requirements, **aerospatiale** has developed a new missile, AS 30 LASER, with laser terminal guidance.

The 240 kg warhead comparable with the remote controlled AS 30's, provides outstanding effectiveness against hardened targets and great penetration capability, due to its high impact velocity.

The firing range (> 10 km) is consistent with optical detection range of most targets.

Terminal laser guidance was selected because of its pinpoint accuracy, and the possibility for the launch aircraft to manoeuvre after missile launch.

The laser designation system can be operated either by the launch-aircraft, or by another aircraft, or by a ground based operator.

For pinpoint interdiction targets, the French Air Force have in their inventory a weapon system consisting of :

- the JAGUAR tactical support aircraft;
- the LASER designation pod (Laser Designation Pod : L.P.D.), developed by THOMSON-CSF for acquisition, automatic target tracking and designation from a single seater aircraft;

- the AS 30 LASER missile, developped by **aerospatiale** (two missiles to be carried per sortie).

3.3. Naval Air Force requirements

Naval targets have very similar characteristics to interdiction ground targets: hardening, surface to air defence systems, good definition.

Their excellent contrast with the environment makes remote detection easier.

Against such targets, the AS 30 LASER is perfectly suitable due to its accuracy, range, and warhead, mostly when high discrimination and/or visual identification are required (crowded and confined areas, such as harbours, channels, fjords, archipelagos).

4. A SPECIFIC APPLICATION BY AEROSPATIALE : EXPERT SYSTEM FOR AS 30 LASER MISSILE LAUNCHER

A missile launcher (ML) is a unit providing the interface between the carrier aircraft and the missile. It allows :

- attaching the missile under the aircraft and transporting it,
- starting up of missile units, transmission of data and firing,
- missile jettisoning in case of emergency.

It consists of a contoured pod including about twenty electronic, electro-mechanical and pyrotechnical systems.

The ML maintenance consists, in case of failure, in locating the faulty unit and replacing it by a spare unit. For this purpose, the ML is tested on an automatic bench which delivers a listing indicating the correct or incorrect test results, and the location of the faulty unit. Location is often affected by indetermination between several elements. This fault location function, which is most important for maintenance, is difficult to control with conventional programming and cannot really evolve (costs and time) to incorporate experience contributions. The system is frozen.

A diagnosis expert system was set up to improve this fault location. The test result listing constitutes the facts entered for consultation, in non-real time as no direct connection exists between the test bench and the expert system.

Installed on an IBM PC XT, this E.S. uses an E.S. generator package using knowledge representation based on production rules.

Programming is performed in a language very close to natural language and is based on 80 rules describing :

- correct operation of the missile launcher,
- cases of failure,
- servicing errors.

The modifications, additions or deletions of production rules are very easy, as opposed to the heavyness of program evolutions in conventional test benches which are frozen systems.

With regard to the results obtained with the conventionally programmed test bench :

- the faulty unit rate of uncertainty has been divided by 3,
- the number of ascertained cases has been multiplied by 3,
- the fault location programming time fora much better result has been divided by 3.

This development evidences the feasibility and interest of E.S. as regards industrial diagnosis. The results obtained and the qualities mentioned hereabove permit envisaging the possibility to use E.S. for Logistics. This is described in paragraph 4 hereafter.

5. E.S. PROSPECTS FOR AVIONICS

As indicated in the previous paragraphs, E.S. can be assumed to be an interesting contribution to Avionics, both from the operational and tool aspect.

The gains which can be expected concern mainly :

5.1. Operation

The diagnosis reliability and velocity and most of all, the indetermination decrease, lead to increase the system availability, which is fundamental for the user.

5.2. Reliability

The importance of handling during removal and refitting operations for maintenance purposes are known to be major elements as regards the Mean Time Between Failures (MTBF). Thus, having a reliable diagnosis, limiting the number of useless removal and refitting operations contributes to a more reliable service of the systems.

5.3. Cost

As a consequence of the two previous gains, the Life Cycle Cost can be improved thru lower rates of unavailability, less unjustified returns for repair, less spare parts stocks due to an improved diagnosis quality. As expertise can be performed by non expert personnel, but using the explanation provided by the E.S., their level should be improved and a gain in personnel expenses can also be expected.

6. CONCLUSIONS

Artificial Intelligence and Expert System currently benefit by a fashion effect in the media which very often mix up (foolish) hopes and crual reality. We tried to show that, although hopes are great and justified, more is still to be achieved in these techniques than has already been done.

Nevertheless, E.S. are no longer a MYTH but a REALITY. E.S's can contribute to Avionics by improving the systems operating condition upholding.

The most important problems to solve or difficulties to overcome are : coherence of the knowledge bases and validation of E.S's.

A more general utilization of this technique and improvement of its performances depend on the improvements in hardware, software and peopleware.

To conclude, the most important thing still to be acquired appears to be the knowledge of the knowledge.

INTEGRATION DU DIALOGUE VOCAL A BORD D'UN AVION DE COMBAT

J.L. BUSTAMANTE

Avions Marcel Dassault - Breguet Aviation
78 Quai Marcel Dassault
92 214 Saint Cloud - FRANCE -

RESUME

L'utilisation d'avions d'armes extrêmement manoeuvrants, dotés de Systèmes de Navigation et d'Attaque complexes permettant de mener à bien des missions variées dans un environnement hostile (vol en zone ennemie à très basse altitude et par tout temps) place le pilote, devenu gestionnaire de la mission, dans des situations où la charge de travail et le stress sont à la limite de ses possibilités.

Dans ce contexte, l'aménagement du poste de pilotage est un élément fondamental dans la réussite de la mission et à ce titre, l'introduction de dispositifs de dialogue vocaux utilisables indépendamment de l'activité manuelle ou visuelle enrichit considérablement l'interface homme-machine et permet l'utilisation des voies orales et auditives jusqu'ici peu sollicitées pour le dialogue avec la machine. Les applications potentielles des dispositifs vocaux concernent, par exemple, l'énoncé explicite des messages d'alarmes en synthèse vocale, les commandes reprenant certaines commandes classiques disponibles en cabine, les commandes permettant par un ordre générique d'effectuer simultanément plusieurs actions, etc.

D'importants travaux ont été menés en France sur le traitement de la parole et, dans le domaine aéronautique, l'appui des Services Officiels a permis d'assurer la transition entre les laboratoires et l'industrie. Ceci s'est concrétisé, dès 1984, par l'expérimentation en vol sur un Mirage d'un boîtier de reconnaissance vocale en mots isolés.

Aujourd'hui, une nouvelle étape est en cours pour l'expérimentation, sur l'avion RAFALE, d'un dispositif de reconnaissance à mots enchaînés.

L'objet de cet exposé est de présenter les travaux réalisés aux Avions Marcel Dassault - Bréguet Aviation, en coopération avec CROUZET et avec le soutien des Services Officiels, le STTE et la DRET, pour assurer l'intégration de ce nouveau moyen de dialogue. *L'essentiel de l'exposé sera relatif à l'utilisation du dialogue* vocal dans une cabine concue dès le départ pour accueillir un tel dispositif. Un outil d'estimation de la charge de travail, destiné à l'analyse des différentes phases de vol avec et sans commande vocale sera présenté.

1. INTRODUCTION

Les pilotes d'avion de combat modernes évoluent dans un environnement très contraignant : facteurs de charges élevés et soutenus, vol à très basse altitude dans un milieu opérationnel à haute densité de menaces, vol tout temps de jour comme de nuit, ... C'est dans ce contexte, que s'effectue la gestion du Système de Navigation et d'Attaque (SNA) dont la complexité s'accroît au fil des années. En conséquence, le pilote est confronté à une charge de travail visuelle et manuelle qui est sans précédent.

De plus, pour réduire la vulnérabilité de l'avion en combat aérien, une manoeuvrabilité très importante est demandée, conduisant ainsi à une prédominance des avions monoplace équipés de sièges fortement inclinés. Cette inclinaison s'accompagne d'une diminution de la surface utile de la planche de bord, limitant ainsi le nombre d'éléments disponibles pour la présentation des informations et pour les diverses commandes. Il est à remarquer que lors de la conduite des phases critiques, l'attention du pilote est concentrée à l'extérieur de la cabine réduisant ainsi sa capacité à gérer pleinement l'acquisition des cibles, l'utilisation des contre-mesures, etc et que le pilotage s'effectue avec "les Mains sur le Manche et la Manette" (concept M3) où de nombreuses commandes ont été regroupées.

D'importants efforts ont été consacrés, aux Avions Marcel Dassault-Breguet Aviation (AMD-BA), à la conception de l'interface pilote-système afin que la charge de travail reste acceptable pour le pilote lors de la réalisation des missions les plus complexes. Ainsi, l'emploi de technologies évoluées comme la visualisation tête haute grand champ à optique diffractive et le viseur-visuel de casque, améliore la présentation des informations lorsque la direction du regard est à l'extérieur de la cabine. De même, la visualisation tête moyenne collimatée, placée sous la visualisation tête haute, autorise une prise d'information plus rapide en évitant l'accomodation visuelle nécessaire avec les visualisations classiques et la planche de bord lors des transitions de la position tête haute à la position tête basse. De nouveaux moyens de commande ont été mis en place pour disposer, à partir d'une surface réduite, des commandes nécessaires pour le dialogue avec le SNA. Il s'agit essentiellement de claviers multiplexés.

Jusqu'à présent, le pilote dispose de moyens nécessitant une attention visuelle et manuelle pour dialoguer avec le SNA. L'introduction du dialogue vocal, c'est à dire de la reconnaissance et de la synthèse vocale, apporte un confort ergonomique très appréciable pour l'introduction de données ou de commandes et pour l'annonce de messages d'alarmes. C'est un moyen de commande supplémentaire, souple d'emploi et aux multiples possibilités, utilisable indépendamment de l'activité manuelle et visuelle, ne demandant aucun aménagement en planche de bord.

2. INCIDENCES DU DIALOGUE VOCAL

Si le dialogue vocal constitue un apport ergonomique important pour l'interface pilote-système, il est cependant indispensable de prendre certaines précautions pour son implantation à bord d'un avion d'armes en considérant, en premier lieu, que c'est un moyen de commande intégré au SNA. Son intégration à bord d'avions existants, essentiellement pour une amélioration des commandes, est relativement délicate et le constat général d'une telle tentative laisse penser que la commande vocale ne peut pas être considérée comme un simple additif aux autres moyens de commande existant.

En effet, même si pour l'instant le dialogue vocal représente un plus ergonomique et opérationnel, il subsiste des commandes classiques équivalentes, en fonction, aux commandes énoncées à la voix. L'incidence de l'introduction d'un Boîtier de Dialogue Vocal à bord d'un avion d'armes concerne la nature même des commandes classiques ainsi que l'intégration des divers éléments ou équipements en relation avec le dialogue vocal.

La compatibilité entre les commandes classiques et la commande vocale doit être assurée par un choix très précis des organes manuels de commande en substituant toutes les commandes stables à libellé fixe (interrupteurs, poussoirs, rotacteurs, claviers, ...) par des éléments instables ou multiplexés. Il est en effet bien difficile de changer l'état d'un interrupteur stable par une commande à la voix !

La richesse de l'application vocale se caractérise par le nombre de fonctions adressables par le dialogue vocal. Pour cela, des échanges d'informations sont nécessaires entre tous les équipements et le dialogue vocal et, par conséquent, l'interconnexion de tous ces éléments au bus avion est indispensable.

L'intégration d'un dispositif de dialogue vocal se traduit pour la cabine par une compatibilité des commandes classiques et des commandes énoncées à la voix et pour "l'avionique" par une interconnexion au bus avion de tous les équipements en relation avec le dialogue vocal.

3. DEMONSTRATEUR RAFALE

Le démonstrateur RAFALE est un avion destiné à tester de nouvelles technologies qui préfigurent celles qui seront utilisées sur les avions de combat de la prochaine décennie. Dès sa conception, l'interface pilote-système a été réalisée avec le souci d'évaluer de nouveaux concepts en matière de visualisation et de commandes. L'application, au démonstrateur RAFALE, de nouvelles philosophies, notamment pour le dialogue avec le système, a permis la réalisation d'une cabine mettant en oeuvre des solutions originales. Parmi celles-ci, figure le dialogue vocal. Avant de développer le type d'application prévu pour l'expérimentation en vol d'un dispositif de dialogue vocal, une présentation de la cabine du démonstrateur RAFALE, du point de vue de l'interface pilote-système, donnera une illustration des règles définies au chapitre précédent.

3.1 PRESENTATION DE LA CABINE

La cabine du démonstrateur RAFALE est conçue autour d'un siège éjectable incliné. Cette installation permet l'exécution de manoeuvres soutenues sous fort facteur de charge tout en réduisant notablement leurs effets physiologiques sur le pilote. L'utilisation d'un siège fortement incliné a fait l'objet d'études approfondies et son influence sur l'organisation de la cabine a été traitée globalement notamment pour l'ensemble des terminaux de visualisation, des commandes, de la tenue du manche et de la manette, de la visibilité extérieure, ...

Le dialogue pilote-système s'appuie essentiellement sur :

- trois terminaux de visualisation,
- un clavier multiplexé,
- un Poste de Modification de Paramètres (PMP),
- des commandes situées sur le manche et la manette des gaz,

3.1.1 TERMINAUX DE VISUALISATION

3.1.1.1 VISUALISATION TETE HAUTE

Les informations nécessaires au pilotage et à la conduite de la mission sont présentées dans la Visualisation Tête Haute (VTH) à optique diffractive. Celle-ci permet la présentation d'informations projetées à l'infini et en superposition avec le paysage extérieur. Les caractéristiques photométriques et la valeur du champ visuel sont bien supérieures à celles des VTH classiques.

3.1.1.2 VISUALISATION TETE MOYENNE

Une Visualisation Tête Moyenne (VTM) collimatée est placée au-dessous de la VTH et en continuité de champ avec celle-ci. Elle est monochrome multimode (tracé cavalier et télévision). Elle présente l'avantage important d'être peu sensible à l'éclairement extérieur. Elle permet la présentation, à l'infini, d'informations relatives à la navigation et aux systèmes avion ainsi que la présentation d'images issues de capteurs et en particulier d'une caméra ventrale, donnant ainsi l'impression d'une planche de bord transparente. Ce concept permet de minimiser le temps de prise en compte des informations lors d'une transition de la position tête haute vers la position tête basse.

3.1.1.3 VISUALISATION TETE LATERALE

Une Visualisation Tête Latérale (VTL) polychrome à tube à rayons cathodiques permet la présentation d'informations relatives à la situation horizontale et aux systèmes avion. L'installation de la VTL sur la droite de la VTH et de la VTM a fait l'objet d'une attention particulière compte-tenu des contraintes liées à l'éjection. En effet, l'installation du pilote sur un siège fortement incliné a pour conséquence de *"relever"* les genoux du pilote au niveau du terminal de visualisation. De plus, les contraintes d'éjection imposent qu'une garde suffisante soit respectée entre le gabarit d'éjection et l'arête horizontale inférieure de la VTL.

3.1.2 COMMANDES

3.1.2.1 CLAVIER MULTIPLEXE

Les commandes pour l'introduction des données relatives aux radio-communications et à la navigation ou pour la sélection de modes ont été regroupées sur un clavier multi-fonctions, limitant ainsi le nombre de postes de commande nécessaires. Celui-ci, situé sous la visualisation tête moyenne, est facilement accessible des deux mains. Il est composé :

- d'un ensemble de touches dédiées servant essentiellement à l'introduction des données numériques,
- d'un afficheur à cristaux liquides permettant la visualisation des paramètres modifiés,
- d'un ensemble de touches, chacune étant associée à un afficheur à cristaux liquides.

3.1.2.2 POSTE DE MODIFICATION DE PARAMETRES

Le Poste de Modification de Paramètres (PMP), implanté sur la planchette de bord gauche permet un accès rapide pour la modification de paramètres. Cet élément est composé d'une couronne externe permettant la sélection du paramètre numérique à modifier et d'un bouton sans fin, coaxial à la couronne. A partir de la valeur courante du paramètre, la modification est effectuée à l'aide du bouton central par incrémentation ou décrémentation. Un bouton poussoir instable permet également de sélectionner ou de désélectionner le calage altimétrique standard. Une fonction annexe de chronomètre figure également avec sa commande. Les paramètres modifiés sont les suivants :

- les canaux des postes V/UHF et UHF,
- le numéro du but de destination,
- le calage altimétrique,
- la valeur du seuil de l'alarme pétrole (Bingo),
- la valeur de la hauteur de garde.

La connexion au bus numérique est assurée par l'intermédiaire du calculateur du clavier multi-fonctions. L'affichage des paramètres modifiés s'effectue sur la visualisation tête haute et dans certains cas, également en VTM ou VTL.

3.1.2.3 COMMANDES SUR MANCHE ET MANETTE

La conception du manche et de la manette des gaz a fait l'objet d'une étude ergonomique poussée tant au niveau de la tenue que de la disposition de toutes les commandes. Il est à remarquer que la manette des gaz est commune aux deux moteurs. Un nombre important de commandes, dont certaines sont multiplexées, équipent le manche et la manette. Parmi celles-ci :

- un joy-stick permet le déplacement d'une alidade sur toute la surface des trois visualisations. La sélection des options ou paramètres s'effectue par appui sur le joy-stick.
- un poussoir de la manette des gaz est utilisé pour la modification de paramètres présentés sur les visualisations.
- un bouton poussoir joue le rôle de tourne pages et permet de sélectionner les diverses pages disponibles en VTL. L'appel des pages peut également s'effectuer par sélection d'étiquettes au joy-stick.

3.2 INTEGRATION DU DIALOGUE VOCAL

L'implantation du dialogue vocal pour l'expérimentation sur le RAFALE à fait l'objet, aux AMD-BA, de nombreux travaux amont effectués avec CROUZET et soutenus par les Services Officiels Français. Parmi ceux-ci, on peut citer les études relatives à la spécification des performances d'un boitier de dialogue vocal à mots enchaînés (soutien du STTE) et les études relatives à l'ergonomie des commandes intégrées manuelles et vocales (soutien de la DRET). L'application présentée ici est issue de ces travaux et sert de base à la définition vocale du RAFALE.

Par ailleurs des vols spécifiques sur un Mirage III sont en cours au Centre d'Essais en Vol (CEV) de Brétigny pour la mise au point des divers algorithmes de reconnaissance.

L'équipement destiné aux expérimentations sur l'avion RAFALE est un boîtier 3/8 ATR réalisé par la société CROUZET et capable d'une synthèse vocale et d'une reconnaissance monolocuteur à mots enchaînés. Il est implanté en soute, à l'arrière du poste de pilotage. Il est connecté au bus numérique et une liaison spécifique, accessible du mécanicien de piste, permet de télécharger les références vocales. En cabine, un alternat dédié au dialogue vocal se situe sur la poignée de manche. Un interrupteur situé sur la banquette droite permet de configurer le Boîtier de Dialogue Vocal en mode apprentissage, vérification ou modification.

3.2.1 REGLES GENERALES

Pour rester homogène avec la définition de la cabine du RAFALE, l'application vocale utilise l'Anglais comme langue aussi bien pour la commande que pour la synthèse vocale. L'emploi d'une langue étrangère constitue une difficulté dans la mesure où le locuteur ne prononce pas les mots de façon constante. Une légère dégradation des performances a été constatée, au sol, pour certains pilotes.

Compte tenu que les différents éléments intervenant dans le dialogue pilote-système peuvent agir de façon indépendante, il a été décidé, dans un but de simplification, de ne pas autoriser un début de modification à la voix et un achèvement par un autre moyen de commande disponible en cabine et vice-versa.

Un retour systématique et sans ambiguïté du résultat de chaque commande énoncée à la voix est réalisé. Ce retour est homogène à celui existant pour les commandes classiques et fait appel essentiellement aux visualisations.

Pour améliorer l'ergonomie des énoncés vocaux, il ne figure pas de mot clé, du type "ENTREE", pour valider ou confirmer une commande. Par contre, le pilote a la possibilité d'annuler la dernière commande vocale émise et, sur demande, d'entendre par synthèse vocale le dernier mot clé reconnu.

Dans les applications développées, la majorité des commandes énoncées à la voix reprend les commmandes classiques déjà existantes. Celles-ci concernent :

- la modification des paramètres modifiables par le Poste de Modification de Paramètres (P.M.P.),
- la désignation des étiquettes situées sur les pages présentées en VTM et en VTB,
- la sélection d'options présentées en VTH, ou en VTM, ou en VTB,
- la sélection du mode approche.

3.2.2 POSTE DE MODIFICATION DE PARAMETRES

L'ensemble des paramètres modifiables au P.M.P est repris à la voix. Cependant, le changement de la fréquence ou du canal des postes UHF et V/UHF s'effectue par le même mot clé. La distinction du canal ou de la fréquence se fait au niveau du nombre de chiffres énoncés.

Par rapport aux commandes classiques, la possibilité de modifier également un but de destination par son indicatif est envisagée. C'est un moyen qui évite la mémorisation du numéro du but. Il est ainsi plus aisé d'énoncer "DEST INDIA TANGO ROMEO", pour la sélection du but relatif à Istres (ITR) ou "DEST ISTRES".

La visualisation de chaque paramètre P.M.P, en cours de modification par la voix, est effectuée sur la visualisation tête haute de façon fugitive.

3.2.3 DIALOGUE AVEC LES VISUALISATIONS

Le dialogue avec les visualisations comprend la désignation des étiquettes situées en bandeau sur les pages présentées en VTM ou en VTB et la sélection d'options permettant entre autres la modification de la valeur de certains paramètres.

L'emploi de la commande vocale permet l'appel direct des pages présentées en VTM ou en VTB. Ces pages sont :

- **En VTB :**
  - la page liste des pannes,
  - la page moteur,
  - la page carburant,
  - la page divers,
  - la page navigation mode ROSE.
- **En VTM :**
  - la page liste des buts,
  - la page way-point,
  - la page recalage.

Quelles que soient les pages présentées en VTM et en VTB, il est possible de sélecter ou de désélecter à la voix les options suivantes :

- le guidage en vitesse sur le but de destination,
- la navigation enchaînée,
- le but auxiliaire,
- un way-point,
- la fréquence VOR.

De même que pour le but de destination, l'appel d'un way-point ou d'un but auxiliaire peut être réalisé à la voix, par son numéro ou par son indicatif. Le même mot clé est utilisé dans les deux cas.

En page navigation en VTB, il est possible à la voix :

- de visualiser et de modifier le but complémentaire par appel d'un numéro de but,
- de modifier le but complémentaire par appel de son indicatif,
- de présenter ou de ne pas présenter la course,
- de modifier la valeur de la course,
- d'afficher ou de ne plus afficher la route désirée du but de destination,
- de visualiser le VOR.

Toutes les options ou paramètres cités ci-dessus ne peuvent être modifiés qu'en page navigation. Si une modification par la voix est demandée sans que la page navigation soit présente, un message vocal spécifie que la page présentée n'est pas la bonne et que par conséquent l'ordre émis ne peut pas être pris en compte.

- **En VTH :**

En tête haute, figurent deux échelles, une pour l'incidence et une pour le facteur de charge. La visualisation ou la non visualisation de chaque échelle peut être demandée à la voix et le même mot clé que celui ayant servi à l'appel de l'échelle est énoncé pour la faire disparaître.

3.2.4 MODE APPROCHE

La sélection ou la désélection du mode approche peut être exécutée à la voix.

3.2.5 SYNTHESE VOCALE

La synthèse vocale est utilisée en permanence pour les retours des diagnostics de reconnaissance et pour les contrôles de cohérence. Les retours de diagnostics sont propres à la reconnaissance vocale et sont du type ; "TOO LOUD", "TOO LOW", "TOO LONG" et "REPEAT".

La notion de contrôle de cohérence est apparue nécessaire au cours des simulations pour indiquer de façon explicite la raison pour laquelle, bien qu'une commande soit correctement reconnue par le BDV, elle n'était pas prise en compte par le système. Les contrôles de cohérence ont pour but de vérifier que la commande énoncée à la voix est compatible à l'instant donné avec l'état du système ou que le format et la structure de l'énoncé sont en accord avec ceux attendus pour cette commande. Un exemple des énoncés des contrôles de cohérence est donné ci-après :

- **"WRONG CHANNEL"** : Si le numéro du canal radio UHF ou V/UHF n'est pas compris entre 1 et 20.
- **"WRONG FREQUENCY"** : Si la fréquence n'est pas une fréquence UHF, V/UHF ou VOR/ILS.
- **"NO POINT"** : Si l'indicatif du but énoncé à la voix ne correspond pas à un but renseigné en mémoire.
- **"MULTI POINTS"** : Si l'indicatif du but est incomplet et qu'il y a ambiguité sur le choix du but.
- **"REPEAT"** : Si le nombre de digits réellement compris est incohérent par rapport au nombre attendu.
- **"WRONG PAGE"** : Si la commande est incompatible avec l'état des visualisations.
- etc.

3.3 ALARMES VOCALES

Dès le premier vol, le RAFALE était équipé d'un boîtier d'alarmes vocales permettant d'énoncer, après un message sonore, un message vocal indiquant de façon explicite l'origine de l'alarme. Une gestion des priorités est effectuée afin d'éviter toute confusion. Lorsque plusieurs alarmes de même priorité apparaissent, les messages vocaux sont entrelacés et sont énoncés jusqu'à ce que le pilote interrompe, de façon volontaire, l'énoncé de ces messages.

4. SIMULATIONS

Une fois l'application vocale définie, des simulations ont été réalisées au centre d'essais des AMD-BA à Istres, sur O.A.S.I.S. (Outil d'Aide à la Spécification d'Informations Système). Ces simulations sont effectuées dans une cabine où l'interface pilote-système est représentative de celle de l'avion RAFALE. Les travaux menés avec CROUZET, soutenus par le STTE et la DRET, ont permis d'investiguer plusieurs domaines et en particulier sur :

- l'accoutumance à la reconnaissance vocale,
- l'apport des commandes énoncées à la voix,
- l'ergonomie des commandes énoncées à la voix.

4.1 ACCOUTUMANCE DES LOCUTEURS

Les diverses simulations effectuées sur O.A.S.I.S. montrent qu'une accoutumance des locuteurs novices est indispensable pour, d'une part mémoriser le vocabulaire et sa syntaxe associée, et d'autre part, pour maitriser l'élocution afin que celle-ci soit compréhensible du dispositif de dialogue vocal. Trois phases distinctes sont identifiées :

- initiation au fonctionnement du dialogue vocal,
- essais,
- fonctionnement opérationnel.

4.1.1 Initiation au fonctionnement du dialogue vocal

Cette phase est effectuée sans couplage à la simulation et permet au pilote de se familiariser avec le fonctionnement de la commande vocale (création des références, élocution, appui alternat...). Quelques essais de reconnaissance sont effectués sur un vocabulaire réduit d'environ 20 mots. Ces essais permettent de trouver l'élocution la plus appropriée pour une bonne reconnaissance, la meilleure étant une élocution naturelle, franche et sans hésitation. Pendant cette phase, les résultats de reconnaissance sont généralement faibles par manque de constance dans l'élocution.

4.1.2 Essais

Lors de cette phase, l'apprentissage complet du vocabulaire de l'application commande vocale est effectué. Le locuteur s'exerce à prononcer les messages aussi naturellement que possible afin d'obtenir des performances acceptables (environ 90 % de taux de reconnaissance). Un essai est composé d'une centaine de phrases syntaxiquement correctes.

Plusieurs itérations sont nécessaires pour dépasser un seuil au-delà duquel il est envisageable de coupler le dialogue vocal avec des simulations, le nombre d'itérations étant fonction du locuteur.

4.1.3 Fonctionnement opérationnel

Le seuil de performance d'environ 90 % étant atteint, les références vocales du pilote sont quasi-définitives. L'application vocale est alors couplée aux simulations de l'avion. Le pilote se familiarise, s'il ne l'est pas, avec l'utilisation des différentes commandes exécutées à la voix et avec leurs effets sur le système (emplacement des retours visuels, accoutumance aux retours par synthèse vocale, configuration des autres commandes, ...).

La représentation suivante donnent une idée de l'évolution du taux de performance en fonction du nombre d'essais et donc de l'accoutumance pour l'ensemble des expérimentateurs.

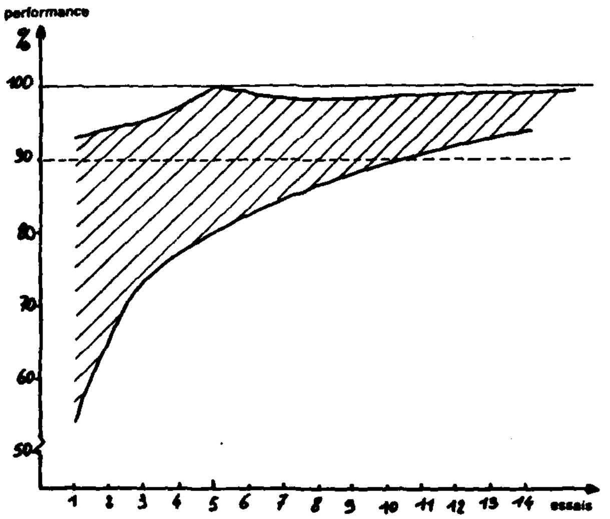


Figure 1. Evolution des performances en fonction de l'accoutumance.

4.2 APPORT DES COMMANDES ENONCEES A LA VOIX

Au cours des simulations, un certain nombre d'avis subjectifs sur l'apport du dialogue vocal, ont été recueillis auprès des pilotes. Ces avis ont été corrélés avec les résultats obtenus par un outil d'évaluation, objectif, visant à donner des informations sur la charge de travail. La mise en oeuvre de cet outil n'est pas destinée à l'obtention d'une valeur quantitative absolue de la charge de travail, liée à l'emploi de telle ou telle commande, compte-tenu du fait que la notion de charge de travail est difficilement appréhendable dans le cadre particulier des simulations effectuées.

L'évaluation de cet outil a été réalisée sur un profil de mission défini à priori et sur lequel de nombreuses commandes devaient être exécutées. Cette mission comporte six segments :

1. décollage et montée à 10 000 ft,
2. palier à 10 000 ft,
3. descente jusqu'à 400 ft,
4. suivi de terrain et panne grave,
5. déroutement,
6. atterrissage sur un terrain de recueil.

La charge de travail est estimée à l'aide deux méthodes similaires faisant appel à une tâche secondaire. Celle-ci consiste à appuyer de façon régulière sur un interrupteur tout en assurant la tâche principale de pilotage. Les deux méthodes d'estimation de la charge de travail, sont dérivées du "Time Estimation Standard Deviation" et du "Tapping Rhythm Task" (1).

4.2.1 TIME ESTIMATION STANDARD DEVIATION

Tout en réalisant la tâche principale de gestion de la mission, le pilote entend de façon cyclique, toutes les 20 secondes, un son d'une durée d'une seconde et de fréquence 750 Hz. Dès que le pilote entend ce son, il effectue un appui sur une pédale située sous son pied droit. Ensuite, tout en continuant la tâche principale, il effectue un deuxième appui sur la pédale, au bout d'un temps estimé de dix secondes. Bien entendu, pendant l'essai toute référence de temps de l'ordre de la seconde est supprimée (pas de montre).

A l'origine, la méthode du Time Estimation Standard Deviation utilise pour l'appui, un bouton situé sur la manette des gaz. Etant donné que pour mener à bien sa mission, le pilote dispose de commandes temps réel situées sur le manche et sur la manette des gaz, l'organe d'appui a été déporté vers les pieds, afin de libérer la main du pilote, au cas où des commandes temps réel seraient nécessaires à l'accomplissement de sa mission.

La méthode consiste à recueillir les écarts de temps par rapport aux 10 secondes estimées et à en calculer l'écart type. Ceci donne une estimation globale de la charge de travail au cours de la mission.

4.3 TAPPING RHYTHM TASK

Cette méthode, très voisine de celle décrite précédemment, consiste à taper aussi régulièrement que possible sur la pédale à un rythme d'un appui toutes les deux secondes. Les écarts entre deux appuis successifs sont analysés et servent à calculer une valeur globale pour le vol. L'évaluation de cette méthode aux Etats Unis, mit en évidence que le rythme était fortement perturbé lors des ajustements de la manette des gaz, le bouton d'appui étant situé sur celle-ci. Des intervalles de temps supérieurs à 5 secondes étaient observés. Comme pour le "Time Estimation Standard Deviation", les simulations réalisées sur OASIS s'effectuent avec l'organe d'appui déporté au niveau du pied droit, évitant ainsi ce problème.

4.3.1 MISE EN OEUVRE DE CES METHODES

Les méthodes du Time Estimation Standard Deviation et du Tapping ne donnent qu'un résultat global de l'estimation de la charge de travail au cours de la mission et ne permettent pas d'identifier les instants critiques lors du déroulement de la mission. Pour cela, des diagrammes sont édités. En particulier un diagramme altitude - temps sert de point de repère rapide et permet d'identifier les différents segments de la mission et un diagramme écart de temps - temps comporte les relevés d'écart de temps par rapport au temps écoulé. Ce dernier permet d'identifier rapidement et précisément les zones où la charge de travail est importante. Ces diagrammes ont l'allure suivante :

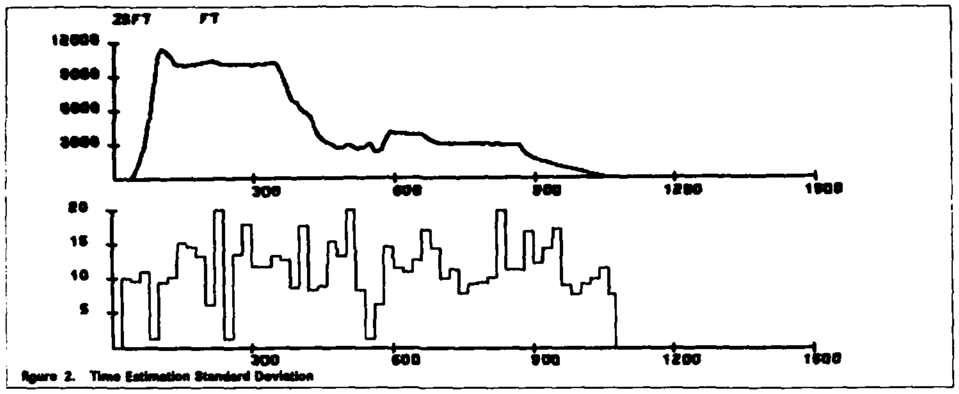


figure 2. Time Estimation Standard Deviation

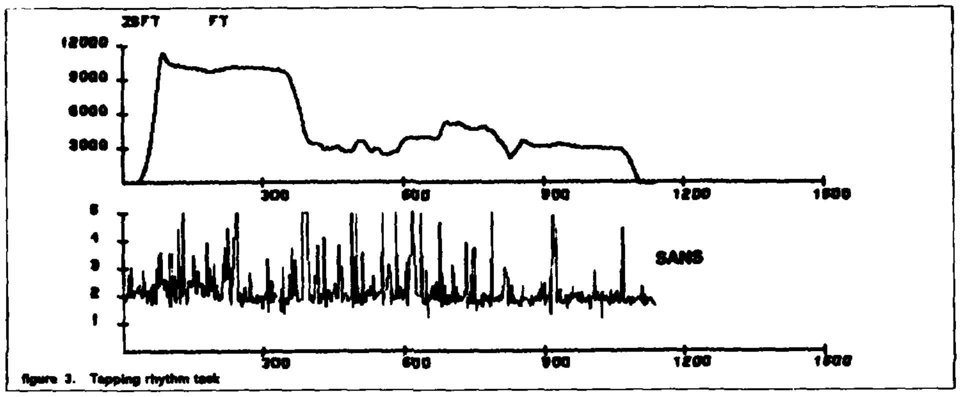


figure 3. Tapping rhythm task

Seule la méthode du Tapping a été retenue après évaluation par des expérimentateurs et des pilotes. Cette méthode est jugée plus souple d'emploi et présente par ailleurs l'avantage de permettre une analyse plus fine des résultats. La majeure partie des simulations a donc été effectuée avec la méthode du Tapping.

Une évaluation pour un expérimentateur donné comporte systématiquement deux simulations, une sans dialogue vocal et une avec dialogue vocal. Les résultats sont ensuite comparés en relatif au niveau des courbes.

Les courbes obtenues font apparaître, dans les deux cas, des valeurs extrêmes mais uniquement maximales. Ceci s'explique par le fait que naturellement, lors de phases critiques, la période des battements augmente. L'écart maximal est alors écrêté à 5 s. Ces extrêmes sont surtout constatés lors des périodes critiques, par exemple lors de la panne survenue en suivi de terrain. L'analyse des diagrammes fait également apparaître des maxima à chaque transition ou manoeuvre délicate, par exemple au moment de la fin de la descente et de l'enclenchement du suivi de terrain. Par ailleurs, compte-tenu de la sensibilité de cette méthode, il s'avère qu'elle ne peut être utilisée efficacement si l'expérimentateur ne maîtrise pas parfaitement l'avion (symbologie, procédures, commandes,...) et la commande vocale. Tous les locuteurs expérimentateurs étaient donc accoutumés à la reconnaissance vocale.

L'examen des courbes avec et sans commande vocale permet de constater que, globalement, l'emploi de la commande vocale est "moins perturbant" que celui des commandes classiques. Ceci est illustré par la comparaison des résultats obtenus par un pilote avec et sans dialogue vocal.

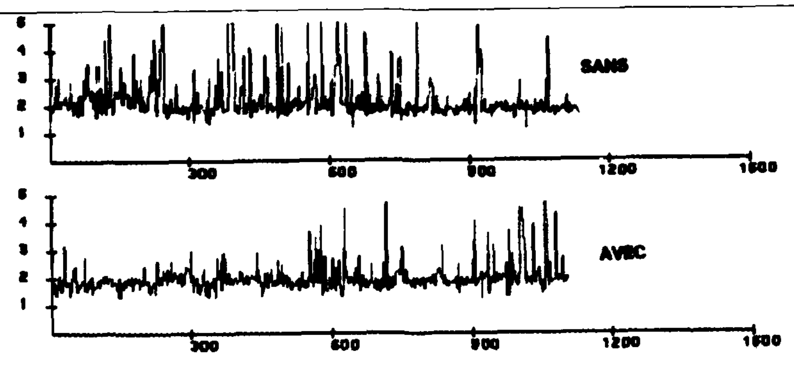


figure 4. Comparaison des diagrammes sans et avec dialogue vocal.

L'analyse des courbes de Tapping permet de constater que des zones de forte perturbation subsistent malgré l'emploi du dialogue vocal, en particulier après la panne et lors de l'atterrissage. Ces points ont fait l'objet d'un complément d'application vocale en cherchant une amélioration du dialogue pilote-système.

L'application vocale développée jusqu'à présent ne fait appel qu'à des commandes dites de "bas niveau". En effet, chaque commande vocale est une reprise pure et simple des commandes classiques existant déjà dans la cabine.

Cependant, le dialogue vocal est un moyen beaucoup plus puissant permettant, par une simple commande générique, de déclencher des procédures connues du système et qui correspondent à de multiples actions élémentaires. Un exemple typique d'une telle commande, dite de "haut niveau", que l'on pense appliquer sur le RAFALE, est la notion de fiches relatives aux terrains d'atterrissage.

Ces fiches sont renseignées au sol et concernent tous les terrains pris en compte lors de la préparation de mission. Des modifications mineures peuvent être apportées en *cours de vol. Tous les paramètres utiles au* système et contenus dans la fiche sont pris en compte lors de l'énoncé d'une commande du type "APPROACH ISTRES" qui sélectionne également le mode approche.

Les commandes de "haut niveau" apportent ainsi une souplesse d'emploi et un gain opérationnnel très appréciable lors de phases de vol nécessitant l'emploi de nombreuses commandes. Ce type de commande est de nature à diminuer très fortement la charge de travail du pilote.

5. CONCLUSIONS

L'introduction de nouvelles technologies dans la cabine d'un avion d'armes comme la visualisation tête haute à optique diffractive, la visualisation tête moyenne collimatée et le dialogue vocal constitue une évolution importante dans la conception de l'interface pilote-système.

Le dialogue vocal destiné aux avions d'armes est un produit de très haute technologie que peu de pays *maîtrisent. A travers le monde, des essais en vol prennent place sur divers types d'appareils afin de préparer* l'introduction d'un tel dispositif sur les avions des prochaines décennies. L'expérimentation du dialogue vocal à bord du RAFALE s'inscrit dans ce cadre là. Bien entendu, des progrès restent à faire dans ce domaine pour atteindre un état où la reconnaissance sera multi-locuteurs avec des performances proches de 100 %.

Néanmoins, l'introduction du dialogue vocal tel qu'il existe aujourd'hui représente un gain opérationnel très appréciable notamment lors de l'utilisation de commandes de haut niveau. Il ne constitue pas, cependant, une solution miracle aux problèmes de l'interface pilote-système et il convient, en particulier, de l'utiliser dans son domaine d'application.

Sur la base des travaux présentés, la définition de la cabine des avions RAFALE des années 95 intègre d'ores et déjà un dispositif de dialogue vocal associé à de nouveaux moyens de commande. Le dialogue vocal jouera un rôle important et permettra de disposer d'une interface pilote-systume adaptée aux mission les plus contraignantes.

6. ABREVIATIONS

AMD-BA :	Avions Marcel Dassault-Breguet Aviation
BDV :	Boltier de Dialogue Vocal
CEV :	Centre d'Essais en Vol
DRET :	Direction des Recherches, Etudes et Techniques
M3 :	Mains sur Manche et Manette
OASIS :	Outil d'Aide à la Spécification d'Informations Système
PMP :	Poste de Modification de Paramètres
SNA :	Système de Navigation et d'Attaque
STTE :	Service Technique des Télécommunications et des Equipements aéronautiques
VTH :	Visualisation Tête Haute
VTM :	Visualisation Tête Moyenne
VTL :	Visualisation Tête Latérale

7. REFERENCE

(1) A comparison of rating scale, secondary-task, physiological, and primary-task workload estimation techniques in a simulated flight task emphasizing communication load. John Casali and Walter W.Wierwielle, Virginia Polytechnic Institute and State University, Blacksburg, Virginia. 1983 Human Factors Society.

INTEGRATION OF ADVANCED SAFETY ENHANCEMENTS FOR F-16 TERRAIN FOLLOWING

James Blaylock
General Dynamics Fort Worth Division
P.O. Box 748
Fort Worth, Texas USA 76101

Donald Swihart
Aeronautical Systems Division
Wright-Patterson Air Force Base
Ohio USA 45433

William Urschel
Aeronautical Systems Division
Wright-Patterson Air Force Base
Ohio USA 45433

Abstract

This paper documents the application of System-Wide Integrity Management (SWIM) to the F-16 Terrain Following (TF) system in order to maximize flight safety during TF operation. The architecture of the F-16 TF system is presented and areas are identified where conventional self-test is not sufficient to ensure flight safety. The concept of system-wide monitoring, or SWIM, is discussed and the SWIM development process used for F-16 TF is reviewed. In addition, the SWIM features developed for the F-16 TF system and the safety benefit and cost effectiveness of these safety enhancements are discussed.

Nomenclature

AFTI	Advanced Fighter Technology Improvements
AMAS	Automated Maneuvering Attack System
Auto-TF	Automatic Terrain Following
BIT	Built-In Test
CADC	Central Air Data Computer
CARA	Combined Altitude Radar Altimeter
C/D	Climb/Dive
CRT	Cyclic Redundancy Test
DFLCC	Digital Flight Control Computer
DFLCS	Digital Flight Control System
FCC	Fire Control Computer
FLIR	Forward Looking Infrared
FMET	Failure Modes Evaluation Testing
GPS	Global Positioning System
HUD	Head-Up Display
INS	Inertial Navigation System
LANTIRN	Low Altitude Navigation and Targeting Infrared for Night
MFD	Multifunction Display
MTF	Manual Terrain Following
OW	Obstacle Warn
RT	Remote Terminal
SCP	Set Clearance Plane
SDC	Signal Data Converter
SMS	Stores Management Set
SWIM	System-Wide Integrity Management
TF	Terrain Following
TFR	Terrain Following Radar
VCW	Vertical Clearance Warning

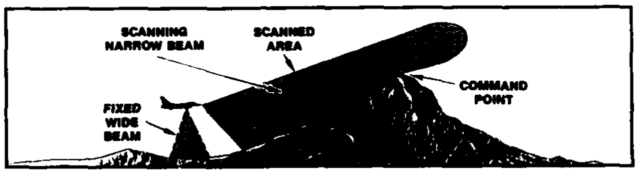
Background

The limited room for on-board equipment in modern fighter aircraft has led to the mechanization of advanced flight-critical functions with nonredundant elements. For the F-16, such space limitations led to the mechanization of a TF system with multiple non-redundant sensors and control processors. In spite of lacking redundancy in critical subsystems, flight safety concerns required that fault tolerance had to be achieved in the overall TF system. As a result, development of the F-16 TF system was accomplished with an advanced flight safety enhancement technique — System-Wide Integrity Management (SWIM). SWIM is a new approach to both the design and utilization of in-flight built-in test (BIT) to detect otherwise undetectable malfunctions that could result in the loss of an aircraft. The SWIM technique was first developed by General Dynamics for the Advanced Fighter Technology Improvements (AFTI) Program for the Automated Maneuvering Attack System (AMAS). The SWIM technique was then applied to the F-16 TF system. Detection of flight-critical malfunctions by SWIM is followed by an automatic recovery maneuver, which consists of a roll to wings-level fly-up for the F-16 TF system.

F-16 TF System Overview

TF System Components

The F-16 TF system provides the capability to fly low level during day, night, or adverse weather conditions at a fixed offset over terrain within aircraft and crew acceleration constraints. Design of the F-16 TF system required the integration of multiple subsystems, as depicted in Figure 1. The TF system generates climb/dive (C/D) commands based on returns from terrain sensors and aircraft state sensors. These commands are then either coupled into the F-16 flight control system for Auto TF or displayed to a pilot for manual terrain following (MTF). The system is comprised of multiple sensors, processors, and displays, as shown in Figure 2. A brief description of the TF subsystems is provided.



- **Forward-Looking Radar Senses Terrain and Associated Computer Generates Climb/Dive Commands**
- **Flight Control System Commands A/C State Change Based on Commands from Radar Computer**
  - **— Direct Coupling in Automatic Terrain Following (TF)**
  - **— Control Stick Coupling of Displayed Commands in Manual TF**
- **Subsystems Feedback Changing A/C State for Command Nulling and Radar Scan Stabilization**
- **Radar Altimeter Controls A/C Over Low Back-Scatter Terrain and Checks Operation During Forward-Looking Radar Control**

Figure 1 Terrain Following Requires Integration of Multiple Subsystems

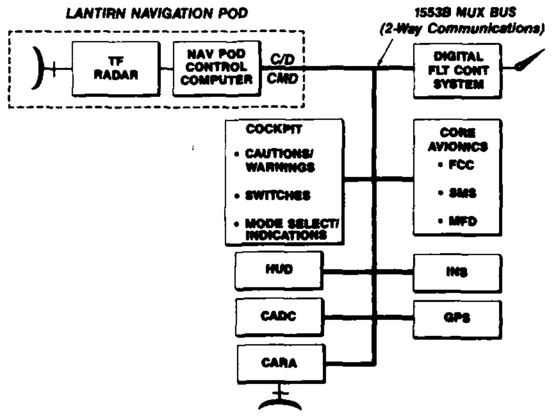


Figure 2 F-16 TF System Architecture

Low Altitude Navigation and Targeting Infrared for Night (LANTIRN) Navigation Pod. The LANTIRN navigation pod consists of two primary subsystems: the forward looking infrared (FLIR) and the terrain-following radar (TFR). The FLIR provides a visible replica of the terrain forward of the aircraft at night. The FLIR is not part of the TF system command loop, but is used as a night aid for pilot-controlled flight. The TFR is the primary sensor for the TF system and provides the computations for the TF system commands. The TFR scans the terrain in front of the aircraft and inputs the scanned terrain range and angle to a command algorithm, which utilizes inputs from other TF subsystems to generate C/D commands.

Inertial Navigation System (INS). The INS provides the platform velocities and coordinates supplied to the navigation pod to stabilize the TFR antenna scan and to calculate the aircraft flight vector for use in C/D command nulling.

Combined Altitude Radar Altimeter (CARA). The down-looking CARA is used to provide C/D commands whenever the terrain reflectivity is too low for the forward-looking TFR to receive sufficient energy for terrain detection, such as over water. CARA is also used as an independent check of the forward-looking TFR operation to detect TFR or INS input angle errors that result in fly-low errors. These fly-low errors would be detected by the CARA as over-flown terrain peaks that were clipped too closely, preventing a potential TFR-controlled clobber. The prevention of a potential TFR-controlled clobber would result since the CARA-based check of TFR-controlled flight compares radar altitude to 75 percent of the TF set clearance with an automatic fly-up manuver being triggered if terrain peaks are approached closer than the 75 percent threshold.

Multifunction Display (MFD). A radio frequency (RF) energy-based confidence display called an E-Squared display is provided on the MFD. This display uses raw TFR video to depict terrain changes in real time. In fog or rain, when FLIR information is unavailable, the pilot can monitor aircraft climb or dive commands with this display.

Head-Up Display (HUD). TF C/D commands are provided on the HUD by the TF command box symbol. The command box is displayed relative to the aircraft flight path vector marker, providing pilot cues for C/D command nulling. The FLIR visual display is provided on the HUD together with the command box.

Central Air Data Computer (CADC). The CADC provides barometric altitude for use in stabilization of INS vertical velocity, which is then used in flight vector calculation for the C/D algorithm.

Fire Control Computer (FCC). The FCC serves as the bus controller between TF subsystems.

Stores Management Set (SMS). The SMS serves as the backup bus controller in the event of an FCC failure.

Global Positioning System (GPS). The GPS provides inertial velocities to compare with INS velocities for INS failure detection.

Digital Flight Control System (DFLCS). The central part of the DFLCS is the quad-redundant Digital Flight Control Computer (DFLCC). The DFLCC interfaces with several flight control subsystems in a redundant fashion, as shown in Figure 3. Failure detection is achieved by multiple cross-channel voting planes implemented in software.

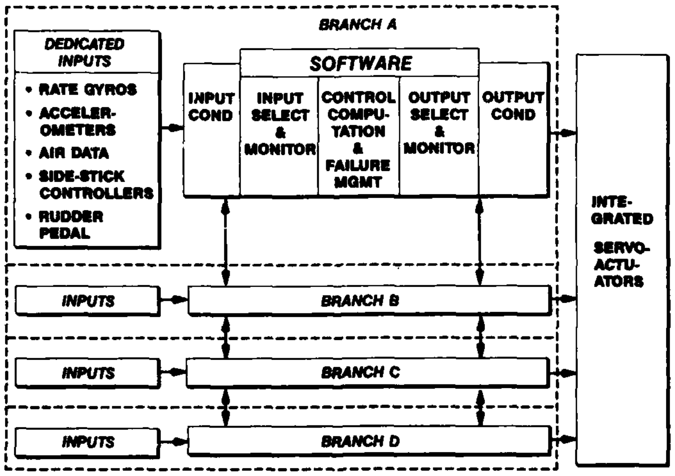


Figure 3 F-16 Digital Flight Control System

All flight-critical flight control sensors are quad redundant and are input to each branch of the DFLCC. Discrete inputs are also connected to each branch. After conversion, the input values are then cross-strapped to each branch through a digital input plane. Monitoring at this point allows sensor miscompares and failures to be identified without failing an entire branch. The data links consist of a serial digital interface controlled by an input/output controller in each branch. Each branch contains one data link transmitter and three receivers. This data link process allows each branch to receive and transmit data to and from the other three branches asynchronously.

Control law outputs are monitored and selected in software by each branch. Since the four branches operate asynchronously, for efficiency Branch B is always used to drive the integrated servoactuators, which in turn drive the control surfaces.

SWIM

SWIM Definition

SWIM, in its simplest definition, is self-test of a function and is system-wide in that it includes all individual subsystems, interfaces, and relevant internal and external influences on the total architecture involved in that function (left circle of Figure 4). The system-wide aspect of SWIM becomes very large if the associated function is of a global nature — as in the case of F-16 flight, which involves all major aircraft subsystems, including engines and flight controls. A reliable communications medium, such as a high-speed multiplex bus, is required among all involved components. With present technology, the concept can be extended to less global functions such as AMAS and TF. SWIM can be thought of as an expanded, traditional (single black box) self-test that operates as a system-level monitor to detect critical subsystem failures and verify system functional integrity (as in the center circle of Figure 4). The management aspect is applied in the SWIM design phase; it involves the elimination of a safety impact associated with a particular subsystem failure rate when that subsystem has minimal involvement in the function (as in the right circle of Figure 4). In other words, the function under consideration, TF in this case, has certain parameters or signals that are safety critical in that undetected aberrations in these signals can result in loss of aircraft. If one of these safety-critical signals is involved minimally in a subsystem or simply routed through that subsystem, then SWIM management involves designing the SWIM monitor architecture to rely on an alternate route or bypass for that signal to ensure detection of failures in that subsystem that affect the critical signal. The safety impact of that particular subsystem is then effectively bypassed. The bypass eliminates safety impact associated with failures of that subsystem since failures of that subsystem are detected by comparison with an alternate source of the same signal.

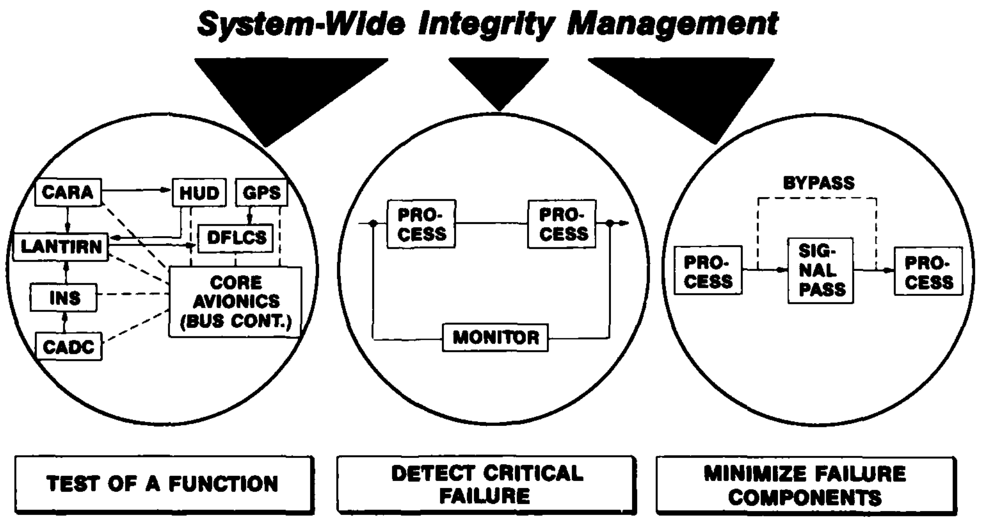


Figure 4 System-Wide Integrity Management

SWIM Development

To develop SWIM for the F-16 TF system, the six-step approach summarized in Figure 5 was followed. The development resulted in the quantification of the safety benefit and the cost of the SWIM features. The goal was to implement only those SWIM features that were cost-effective in reducing the predicted mishap rate for F-16 TF.

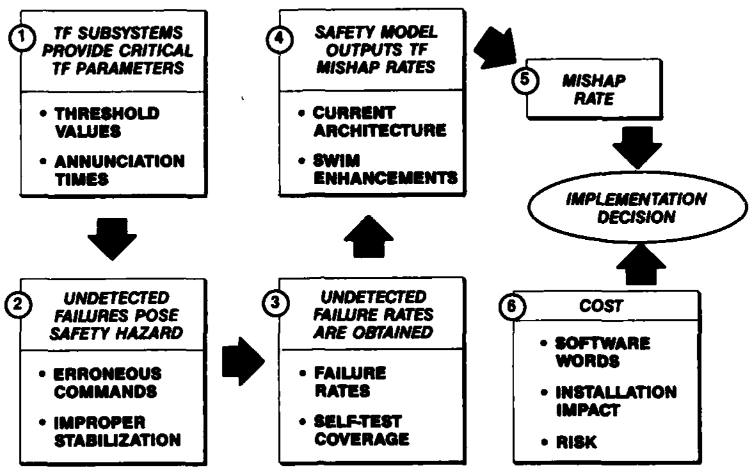


Figure 5 TF Swim Development Approach

The first step in the SWIM development was to determine the signals critical to TF, the maximum error threshold values of these signals that could be tolerated without safety impact, and the maximum tolerable delay before annunciation of these threshold-exceeding critical signals. The second step was defining the consequences of undetected critical signal failures, and the third step was to determine the degree of safety provided by the existing self-test provisions of the TF subsystems. One output of this analysis was the rate of occurrence of undetected hazardous failures. This rate was estimated by a variety of analysis techniques, ranging from detailed piece-part failure modes evaluations to the analysis of system-level failure effects. In the fourth step, an aircraft safety analysis was performed to predict a baseline aircraft mishap rate resulting from undetected hazardous failures. The safety analysis was accomplished using a fault tree analysis methodology, which models many effects such as operational usage and pilot response to failure modes in order to obtain the most realistic prediction of mishap rates practicable. Once individual undetected failure modes were determined, SWIM features were mechanized to provide detection capability. Next, the aircraft safety analysis was again conducted to determine the mishap rate with SWIM features. In the fifth step, the mishap rate with the SWIM features was compared with the baseline mishap rate to obtain a measure of the benefit of the SWIM features. In the sixth step, the rough design for the SWIM features was used to estimate a cost for the features. The cost consisted of several factors — including schedule impact, estimates of software change size, and risk associated with development. For example, the attitude estimator required 150 software words in the DFLCC, had demonstrated low risk based on previous AMAS use, and could be implemented relatively quickly. The decision as to which SWIM safety enhancements to implement was then made by the Air Force based on projected number of aircraft saved and cost to implement the SWIM feature. The complement of SWIM features selected for F-16 TF required approximately 2750 software words at a cost of less than $2 million to achieve a predicted aircraft savings of approximately $200 million over a 20-year life of 350 LANTIRN-equipped F-16 aircraft.

SWIM Analysis Results

SWIM Monitors

Each subsystem involved in the generation of TF commands was evaluated using the SWIM analysis methodology. It was concluded that, although the individual TF subsystems have a relatively high internal self-test coverage, areas exist in which significant improvements in the overall TF system self-test coverage could be achieved. Although it was not feasible to develop monitors for every undetected failure mode identified by the analysis, monitors were developed for the failure modes that contributed the major portion of the undetected failure mode mishap rate. The SWIM monitor features required a redundant, cross-checked host with enough reserve capacity to implement the features in order to have guaranteed integrity of the SWIM features. The quad-redundant DFLCC provided the required host for the SWIM features. In Figure 6, the F-16 TF SWIM features are listed in terms of the basic undetected malfunction causes that they were designed to cover. A brief description of the critical undetected failure mode areas and the SWIM features designed to cover them follows.

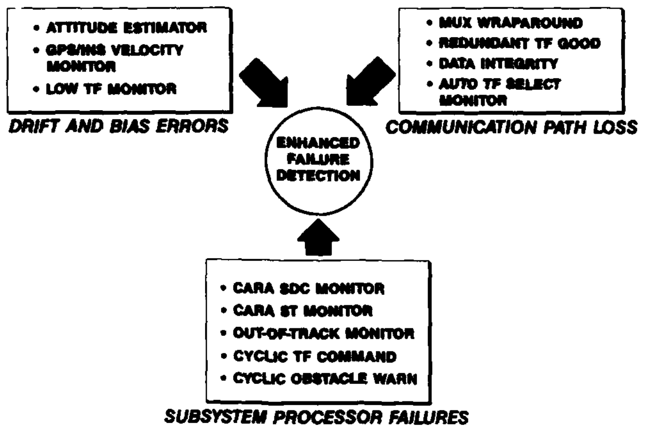


Figure 6 F-16 TF Swim Features . . . Cover Different Undetected Malfunctions

Sensor Drift and Bias Errors. Table 1 contains the SWIM features designed to cover hazardous drift or bias error malfunctions of the INS or TFR. Data from both the INS and the TFR can be corrupted by drift and bias errors that are not detectable by subsystem self-tests. In the case of the INS, monitors were implemented that could detect these errors using independent motion reference sources. The GPS provides a highly accurate source of velocity data for the GPS/INS velocity monitor, while the flight control gyros provide a source for the attitude estimator to calculate a short-term estimate of aircraft attitude. No additional source of forward-looking radar data is available for comparing with the TFR data, but data from CARA is available to provide vertical plane monitoring of nearness to terrain peaks during TFR-controlled flight with the low TF monitor for detection of fly-low angle error effects. If the CARA altitude becomes less than 75 percent of the TF set clearance, an automatic fly-up maneuver is initiated based either on a similar low-altitude check circuit in the LANTIRN navigation pod or by the independent DFLCS low TF monitor.

Table 1 Drift and Bias Errors Swim Features

FEATURE	FAILURE DETECTION	LOCATION
ATTITUDE ESTIMATOR	INS ATTITUDE	DFLCS
GPS/INS VELOCITY MONITOR	INS INERTIAL VELOCITY	DFLCS
LOW TF MONITOR	TFR FLY-LOW ERROR	DFLCS

Subsystem Processor Self-Test Coverage. Table 2 is a summary of the SWIM features covering subsystem processor hazardous malfunctions. Additional coverage is provided for undetected failures in the CARA signal data converter (SDC) via the CARA SDC monitor since the SDC does not have significant coverage. The CARA receiver/transmitter coverage is significant because of automatic self-test and out-of-track detection circuitry, both of which are monitored by DFLCS SWIM features. Failures in the CARA SDC are monitored by comparing the analog CARA data presented to the SDC with the digitized radar data the SDC provides to the TF system. Relatively straightforward monitors could not be implemented for the TFR processors; instead, a cyclic redundancy test (CRT) was used to implement a cyclic TF command and obstacle warn algorithm problem. The cyclic problems involve comparing computational test cases performed in the TFR processors with predetermined results for those test cases.

Swim Features

FEATURE	FAILURE DETECTION	LOCATION
CARA SDC MONITOR	SDC	DFLCS
CARA SELF-TEST MONITOR	ALTITUDE ABOVE GROUND LEVEL	DFLCS
OUT-OF-TRACK MONITOR	ALTITUDE ABOVE GROUND LEVEL	DFLCS
CYCLIC TF COMMAND PROBLEM	COMMAND PROCESSOR	DFLCS/NAV POD
CYCLIC OBSTACLE WARN TEST	OBSTACLE WARN PROCESSOR	DFLCS/NAV POD

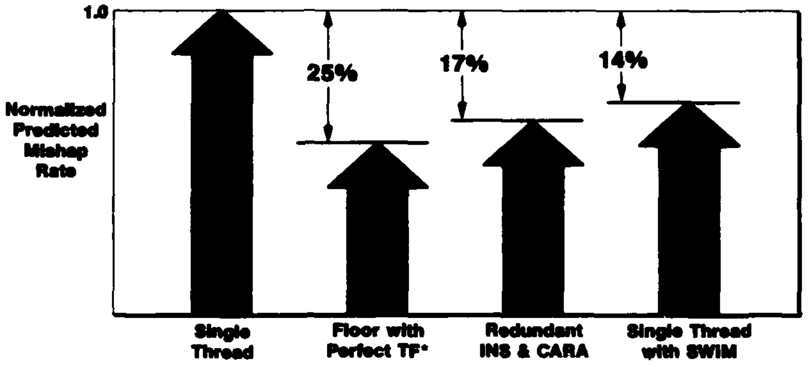
System Communication Paths. Table 3 contains the SWIM features implemented to prevent undetected loss of communication paths. Undetected failure modes exist in the 1553 data bus remote terminals (RT) of various TF subsystems. Several methods are used to monitor these failures: for the CARA RT, the CARA SDC monitor provides sufficient coverage of TF failures; wraparound checks of communication paths with alternating data values are used to monitor the other RT failure modes. Redundant TF data good and data integrity discretes provided by subsystem processors are also monitored to identify invalid TF operation. These discretes are external to the 1553 communication bus. An Auto TF select monitor verifies Auto TF engagement to prevent the pilot from relinquishing vertical clearance control to a disengaged Auto TF system.

Table 3 Communication Path Loss Swim Features

FEATURE	FAILURE DETECTION	LOCATION
MUX WRAPAROUND	TF INFO TRANSMISSION	DFLCS/NAV POD
REDUNDANT TF GOOD	NAV POD TF STATUS LOSS	DFLCS
DATA INTEGRITY	TF SUBSYSTEM STATUS LOSS	DFLCS
AUTO TF SELECT MONITOR	AUTO TF DISENGAGEMENT	DFLCS

Predicted Mishap Rate Improvement

In Figure 7, the percent reduction in mishap rate is depicted for various F-16 TF configurations; single-thread, redundant (dual CARA and INS), and single-thread with SWIM. As can be seen from Figure 7, redundancy affords maximum safety benefit at 17 percent reduction in predicted mishap rate. However, the addition of SWIM to the single-thread configuration is seen to achieve 14 percent predicted mishap rate reduction at far less cost and installation impact than the redundant configuration. It should be noted that the mishap rate improvement in the low-altitude, high-speed regime is limited by factors such as engine malfunction and bird strike — as depicted by the fact that only 25 percent improvement is achievable with a perfect TF system.

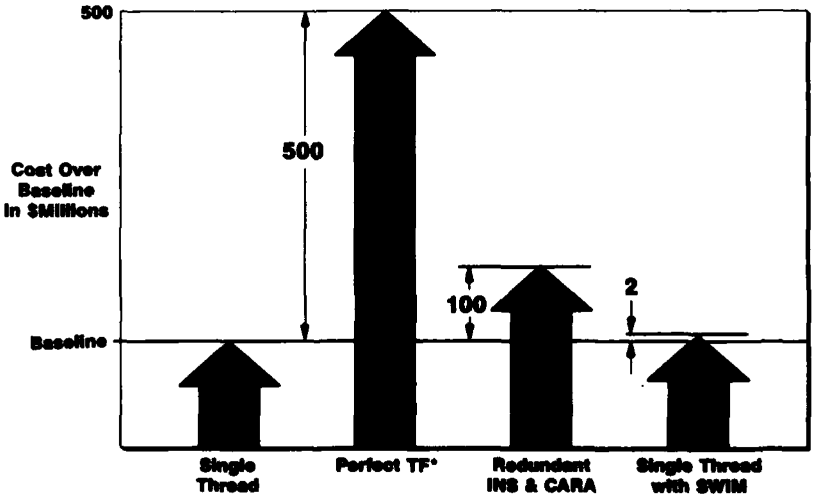


** Mishap Rate Improvement in High-Speed, Low-Altitude Regime Limited by Factors Such as Engine Malfunction and Bird Strike*

Figure 7 F-16 TF Predicted Mishap Rate Reduction

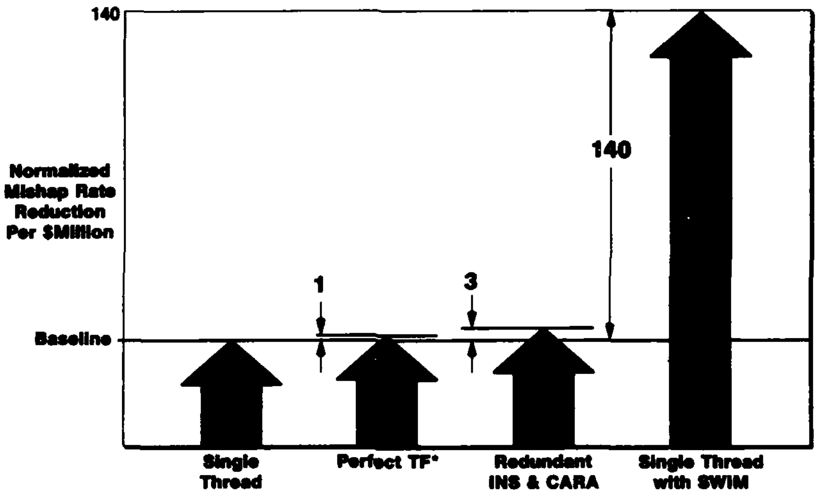
Cost Effectiveness

As previously indicated, the cost to add SWIM to the single-thread baseline F-16 TF system was $2 million. This cost and the additional cost over the single-thread baseline of a redundant INS and CARA TF system and of a perfect TF system are compared in Figure 8. The costs depicted in Figure 8 are all based on incorporation into 350 LANTIRN-equipped F-16 aircraft. As shown in Figure 8, a cost of approximately $500 million is required to achieve the perfect TF system, in terms of hazardous malfunction detection, via triplex subsystems, including the TF sensor. Of more interest is the cost of more traditional TF system redundancy involving at least dual subsystems except for the single TF sensor. The traditional redundant TF system costs approximately $100 million to achieve the same order of aircraft savings as single-thread with SWIM at $2 million. From the Figure 7 mishap rate reduction data and the Figure 8 cost data, a cost-effectiveness comparison can be made for the alternate F-16 TF systems relative to the single-thread TF baseline. Figure 9 is a graphic illustration of the cost-effectiveness comparisons from which it can be seen that — in terms of safety improvement — the application of SWIM to the single-thread configuration resulted in 140 times as cost-effective a TF system as a perfect system and 47 times (140/3) as cost-effective a system as the redundant INS and CARA system.



** Detection of virtually all hazardous malfunctions utilizing triplex INS, CARA, AND TF sensor*

Figure 8 Alternative F-16 TF System Cost



* Detection of virtually all hazardous malfunctions utilizing triplex INS, CARA, and TF sensor

Figure 9 Alternate F-16 TF System Cost Effectiveness

Monitor Implementation

It is necessary to implement the monitors in a reliable fault-tolerant computational system to ensure reliability of the SWIM monitor circuitry and also prevent erroneous failure declarations. The DFLCC is an ideal home for the SWIM monitors, because the fault-tolerant architecture for such processing already existed to handle the failure management needs of the flight control system. The monitors were implemented as an additional software function in each redundant branch of the quad DFLCC. The impact to the Flight Control Computer Operational Flight Program and throughput is small, as only the SWIM monitor logic was required to be added. The existing software to implement signal and failure management was utilized to ensure the SWIM monitors were not affected by failures in the flight control computer.

In addition to the SWIM monitors already discussed, additional monitors were implemented in flight control software to assess the health of the various TF subsystem components. The status of each subsystem BIT is reported to the DFLCC, and monitors are implemented to determine the health of each subsystem based on the subsystem BIT status. The DFLCS also checks data passed between the TF subsystems to evaluate subsystem health. The checks verify that the data values are within expected bounds.

Monitor Action

Whenever a failure in the TF system is detected, the method used to correct the failure is to terminate TF and provide an automatic roll to wings-level fly-up. The incremental-g level for a particular fly-up depends on the system or condition that detected the failure. Table 4 lists the incremental-g levels and the system initiating the fly-up commands.

The subsystem failures are detected by self-test or by a SWIM monitor. The Vertical Clearance Warning (VCW) is based on CARA altitude and is caused by the airplane penetrating 75 percent of the SCP. The Obstacle Warn (OW) is based on the TFR and is generated when some object not previously sensed by the TFR is detected within an approximately 3-g template in front of the airplane.

Table 4 Automtic Roll to Wings-Level Fly-Up G Levels

INCREMENTAL G	SYSTEM
2	Subsystem Self-Test or SWIM
3	Vertical Clearance Warning
4	Obstacle Warn

System Validation

An extensive failure modes evaluation testing (FMET) was conducted, in addition to the standard validation tests, to establish confidence in SWIM monitor operation in the face of actual or simulated hardware failures and to ensure that proper pilot displays and handling qualities are maintained during failure identification and recovery. FMET was conducted in real time in a handling qualities simulator with a pilot in the loop. Critical TF system processor hardware, such as the avionics suite, the flight control computer, and the TFR processors, were included to ensure correct timing and processing of the TF functions. In addition, all of the selected SWIM features are being evaluated in F-16/LANTIRN TF flight tests, which began in January of 1988 and are scheduled to continue until late 1988.

SWIM Study Conclusions

For optimum failure detection in any system, redundancy is the best implementation method, as redundancy provides a means to detect failures by comparison. However, as previously discussed, redundancy is not present in many of the F-16 TF subsystem sensors or processors due to space limitations.

The SWIM method of failure detection was developed to enhance hazardous failure detection in a single-thread system. This method was not intended to replace redundancy. However, System-Wide Integrity Management did accomplish its intended purpose — a significant decrease in predicted mishaps from otherwise undetected failures in the TF system. In addition, SWIM achieved almost as much predicted mishap rate improvement as the redundant system at far less cost and installation impact for a phenomenal cost-effectiveness advantage over traditional redundancy.

DISCUSSION

J.Bart, US

What was the False Alarm Rate (Improper Identification of Failure Condition) using the SWIM approach?

Author's Reply

The implementation of either a digital or software implemented monitor could have an initially unacceptable false alarm rate due to too low a value of persistence counts or too tight trip levels. However, these values are generally corrected during system simulation and flight test and achieve an acceptable false alarm rate. However, software monitors utilizing existing hardware have an inherent advantage over traditional hardware monitors, in that they do not suffer from the additional false alarm rate associated with hardware self test and component drift errors. For SWIM, the implementation of the monitors in software has thus resulted in a quantitatively lower false alarm rate than systems implemented with traditional redundant hardware techniques.

CAN Ada (MIL-STD-1815A) FLY MISSILES?

by
Carl W. Hall
Naval Weapons Center
Missile Software Branch
China Lake, California USA 93555-6001

SUMMARY

A small group of software engineers at the Naval Weapons Center explored the transition of the MIL-STD-1815A (Ada) language and associated technology into the Navy missile domain beginning in October 1984. The project was funded by Independent Exploratory Development (IED) funds. It had a principal objective to develop operational software for a typical navy missile in Ada, while examining the issues associated with deployment of the Department of Defense (DoD) standard language in a real-time missile environment.

The project lasted 3 years and was finished in September 1987 with the Ada software running in a breadboard version of the missile guidance computer. The final version of the software had almost all the missile functions included with the exception of the safety module. Safety deals with the range requirements during testing and this data were not available. The computer and software are being moved to the Hardware-In-the-Loop (HWIL) facility where they will be integrated with various missile hardware and a six-degree-of-freedom simulation. Current plans are to seek funding to fly a test missile at the Naval Weapons Center range with a modified version of the IED Ada code.

BACKGROUND

Ada is a computer language, the product of a 12-year development program designed to provide a standardized and effective higher-order programming language for DoD use. Its features and constructs were designed to supply a tool to control software costs for tactical embedded computer systems in military weapon systems. In July 1983, the DoD Deputy Director for Research and Engineering issued a policy directive that read in part, "Effective 1 January 1984, for programs entering Advanced Development, and 1 July 1984, for programs entering Full-Scale Engineering Development (FSED), Ada shall be the programming language."

The North Atlantic Treaty Organization (NATO) committed to use Ada for the implementation of its information systems that support the exercise of command, control, and communication within the Alliance. On 25 January 1985, the NATO Defense Planning Committee approved NATO's Ada Policy, effective in 1986. The Policy made Ada the single higher-order language for all NATO command and control information system projects not involving France. NATO permits the use of another language for such systems if a waiver is requested and granted.

Ada became an international standard on 12 March 1987 when the Central Secretariat of the International Standards Organization (ISO) announced unanimous approval of the Draft International Standard (DIS) 8652—Programming Language Ada. The standard is known as ISO/8652-1987 Programming Language Ada and is an endorsement of the American and French Ada standards (ANS/MIL-STD-1815A and NF Z 65-700, respectively).

In March and April of 1987, the United States Deputy Secretary of Defense (the Honorable William H. Taft, IV) gave further support to the Ada software initiative by signing two new directives (3405.1 and 3405.2). Directive 3405.1, the "Computer Programming Language Policy," states "The Ada programming language shall be single, common, computer programming language for Defense computer resources used in intelligence systems, for the command and control of military forces, or as an integral part of a weapon system...Ada shall be used for all other applications, except when the use of another approved higher order language is more cost effective over the application's life-cycle, in keeping with the long-range goal of establishing Ada as the primary DoD higher order language."

Directive 3405.2, the "Use of Ada in Weapon System," states that Ada shall be the single, common, higher-order language used for all new weapon systems during all phases of the life cycle and to major software upgrades of existing weapon systems. The directive further requires the use of validated Ada compilers, of software engineering principles described in DOD-STD-2167 and DOD-HDBK-281, and of Ada as a program design language (PDL) to design software.

INTRODUCTION

This exploratory development program attempted to analyze the major issues associated with inserting Ada in a missile. In order to achieve the goals of this project, it was deemed necessary to avoid the popular trend of translating a working program into Ada. Avoiding translation of an existing program meant the important issues involved in utilizing Ada in all the life cycle phases could be studied. The power of Ada is sometimes lost when translating from a different language into Ada because the programmer is not anxious to rebuild a working design. Another danger is the tendency to program in Ada as one does in other languages. A good FORTRAN design may not be a good Ada design. In fact, there is evidence to the contrary.

Therefore, a missile program was selected in which the missile had not yet been flown and for which little or no code had been written. The specifications for the missile, the weapon's tactical computer design, and the computer program performance specification (CPPS) were available and could be used to develop the operational flight software (OFS) in Ada. A breadboard version of the guidance computer was built for the HWIL facility from the contractor's design and could be used to test the OFS once completed.

During the development, which spanned 3 years, controls had to be established to avoid losing sight of the objectives. These controls were specified in a software development plan (SDP). Frequent peer group reviews were held. Also each contributor had to maintain a programmer/analyst notebook, which was reviewed periodically by the principal investigator of the project. Instead of developing a software quality evaluation plan, the process was included in the SDP.

In the SDP, the major issues were listed in order to maintain focus on what the project's objectives were. These issues changed through the course of the project as circumstances and funding levels changed. The following list provides the final state of those issues after several iterations had occurred during the course of the project:

1. How suitable are the Ada features for a stringent real-time problem with limited throughput and memory such as missiles have?

2. What are the theoretical and pragmatic issues one must solve or understand to implement Ada in an embedded tactical weapon system?

3. What issues are associated with the compiler, runtime system (RTS), tools, and compiler validation?

4. Can useful software metrics be automated that measure Ada's effectiveness and the programmer's final Ada product?

5. Can Ada benchmarks or performance tests help quantify a validated Ada compiler's time and memory expansion factors over assembly language?

6. Does Ada improve productivity, maintainability, and reliability of weapon software while reducing cost and development time?

As the project progressed, the decision to change the guidance computer microprocessor from a Zilog Z8002 to a Motorola MC68020 gave the IED task a new dimension in its insertion effort. The theory of Ada's transportability or machine independence came into the pragmatic world of a new compiler, a new development system, cost overrun, schedule slippage, a new emulator, and procurement cycle delays. The particular Ada compiler purchased required a Digital Equipment Corporation MicroVax II computer. The first emulator purchased never functioned as advertised and had to be abandoned after 6 months of frustration.

Nevertheless, the project concluded on schedule and under cost with the OFS coded in Ada and running open-looped in the missile guidance computer. In September 1987, there were about 222,000 bytes of compiled Ada object code and approximately 11,000 lines of Ada statements in the missile program. Whereas some issues remain under investigation, others were examined and concluded.

DESIGN METHODOLOGY

Several methods were studied before selecting one with which to design the IED software. The popular method being touted for Ada programming was the object-oriented design (OOD) methodology [1]. Structured analysis/structured design (SA/SD) [2] was a popular design method of the 1970s. The object flow method used object flow graphs instead of data flow diagrams (DFDs). Other less used techniques were the State Machine approach and Petri Nets. Traditional mathematical flowcharts are good for illustrating the algorithmic flow of the program but tend to be written only after a program is working.

Because the computer program performance specification was functionally decomposed, it seemed at the time reasonable to assume that SA/SD methodology would map better onto the missile domain selected. The priority and sequencing of missile tasks were very rigid in most modes. Also, the computer only had a single microprocessor, which implied linear processing, and preliminary investigation had flagged the Ada tasking rendezvous as a concern. This choice of SA/SD caused some problems during the project. DFDs and structure charts do not handle interrupts nor illustrate control very well. The data dictionary is an excellent resource, but it lacks specific information required to type the data objects.

Because initial discussions with compiler vendors established large rendezvous times exceeding 1 millisecond (some were as high as 10 milliseconds), plans were made to build a generic executive (GENE) to operate cyclically on a time-slice theory. The intent was to use as much of the runtime system as possible and allow GENE to handle the "hot" spots like tasking and interrupts. This succeeded quite well. In May 1987, the Software Engineering Institute at Carnegie-Mellon University published a technical report [3] which stated on page 43, "Most of Ada's benefits can be achieved for real-time systems even if Ada's tasking features are not used...If Ada tasks are not used,...the need to use an existing real-time executive or write a special-purpose executive must be included as part of any project plan."

The concept for the IED executive was quite simple in structure as illustrated in Figure 1. The executive isolates the application tasks from both the runtime system and the computer with its interrupts and Input/Output operations. The ideal situation, however, is when the runtime system can handle the missile environment efficiently enough to not require a complementary cyclic executive like GENE. The reason its transportability. If one is not careful, the cyclic executive becomes so specialized that it cannot be used on other projects.

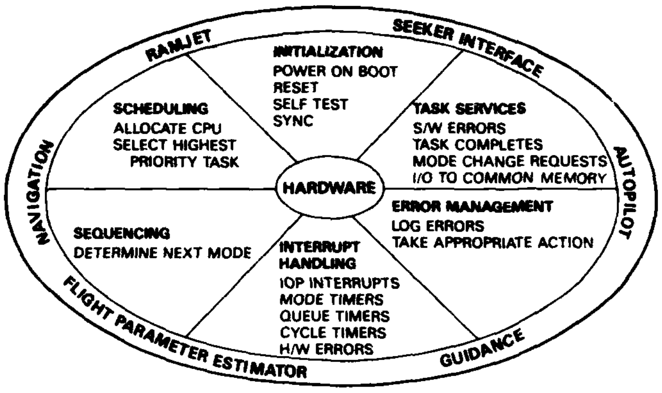


FIGURE 1. GENE.

To avoid this problem, GENE was broken into two components; viz., generic and primitive. The generic component was designed to work with other missile projects. GENE accomplished this and is in another missile program operating efficiently and correctly. The generic component is reusable, however, if both applications use the same language. The primitive component is machine-dependent and will not transport. It is specifically tailored to optimize the communication between the machine and the generic component. In most instances, this means programming in assembly language. For the IED code, GENE was programmed in Ada and its primitives in assembly language.

GENE/RTS was at the heart of the IED software. The basic top-level structure was the introduction of an application package. GENE/RTS could theoretically invoke different application packages for different missiles. Because external missile sensors had direct memory access, a global data package was created. However, the original intent was to place only type declarations and constants in a global package available to all the application tasks. Each major task was designed to use parameter calls with the Ada in, in out, out mechanism with its subordinate procedures. For clarity, the major tasks had to include Input/Output lists in their respective "module header." In this way, each task could be invoked when an interrupt occurred, and if data were ready in the global package, it would be autonomous from the other missile tasks. When data was not ready, GENE kept the task asleep until the interrupt occurred.

The last design technique applied was the use of Ada as a PDL. After attending several demonstrations of Ada superset PDL tools like BYRON built by Rational Inc., a decision was made to use Ada only. The fundamental reason for rejecting an Ada superset was cost. The IED project was on an austere budget and couldn't afford the expense. Therefore, developers of the various Ada tasks were directed to use Ada as a programming design language until their top-level design was completed and reviewed. Each design was reviewed in a structured walkthrough as outlined in reference [4].

One further note concerning the IED software design process. The belief that one particular design methodology will solve all the design issues is not in harmony with practical experience. A project should use a disciplined approach to the development of software and build early prototypes to test these preliminary designs. Most successful software engineers have been doing this for years even though DoD Directive 5000.29 essentially forbids it. The IED team discovered that the missile CPPS was incomplete and that the requirements were constantly in motion. The navigation prototype helped establish the missing specifications to a large degree. Figure 2 shows the DOD-STD-2167 waterfall, the defense software procurement cycle, and the operational software and shows how the prototype software relates to these three life cycle categories.

COMPILER

The project began with a nonvalidated Ada compiler for the Z8002. There was no validated compiler at the time. The first lesson learned was NEVER use a nonvalidated compiler. One frequently did not know when errors occurred whether they were actual errors or the compiler's problem. Also, all the workarounds are time-consuming and demoralizing. The commercial company building the compiler promised imminent validation for 2 years. Today's market of validated Ada compilers is large enough that this lesson shouldn't be relearned by anyone.

When the decision was made to replace the Zilog Z8002 microprocessor in the guidance computer with a Motorola MC68020 and MC68881, a search began for another Ada compiler. This effort spawned a task to develop some formal method for evaluating Ada compilers built for a desired processor. This effort was documented in reference [5]. Kernel to the task was the generation or inheritance of 18,000 lines of Ada benchmarks. These benchmarks were used more like performance tests. The Ada Joint Program Office (AJPO) in the Office of the Secretary of Defense (OSD), responding to the fact that validation does not quantify performance, has a contract in place to procure an Ada Compiler Evaluation Capability. This contract is administered by Wright-Patterson Air Force Base and performed by the Boeing Company.

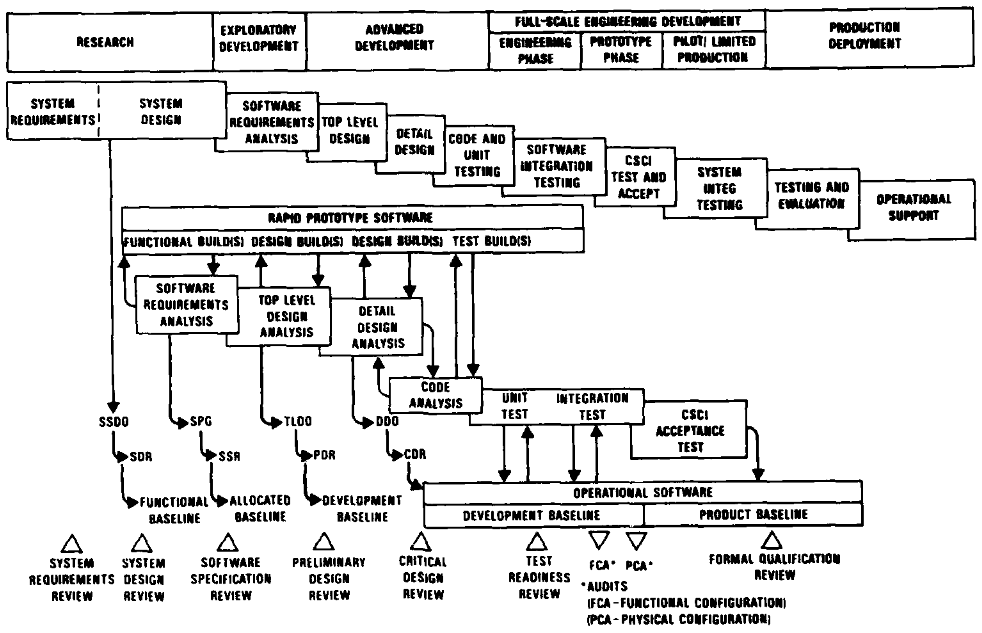


FIGURE 2. Rapid Prototype Software.

Although performance of an Ada compiler is critical in real-time systems, one should consider other criteria when purchasing an Ada compiler. The following list summarizes some of the criteria:

1. Performance of both the object code and the runtime system
2. Speed of compiler from compiling to downloading (compile, link, load)
3. Development environment
4. Cost
5. Host computer and target
6. Emulator (if target is a microprocessor)
7. Symbolic debugger (on-line debugging capability)
8. Diagnostics in compiler (compilation and runtime)
9. Security
10. Language Reference Manual, Chapter 13 implementation and Appendix B (Pragma Interface and In-line, address clauses, etc.)
11. Vendor product support

CONCLUSION

The project was documented in two reports. The summary report is entitled *Independent Exploratory Development Ada Demonstration Project* [6], and the other is *IED Ada Demonstration Project Software Development Plan* [7]. There are volumes of design data, data flow diagrams, data dictionaries, and structure charts documenting the design and structure of the software. The test results are incomplete until the HWIL testing is concluded.

The productivity figures for this effort showed a definite increase of between two and five times over comparable assembly language projects. The variance in these figures is due to the fact that the project did not have an assembly language version of the software for comparison. Industry standards range from a bidding data base of about five documented validated verified lines of code (DVVL) to a cottage industry figure of around 12 DVVL. Comparable cost factors showed a decrease in cost of between 15 and 10 times.

In analyzing these two results, there are perhaps factors that influenced those numbers external to the Ada language itself. The project was a research study and was staffed with high-caliber people. A program performance specification was available at the beginning of the project. Although it was not complete or 100% accurate, it was tremendously useful. This is unlike many projects where the specifications and requirements evolve during development. Also, the documentation requirements were not the entire DOD-STD-2167 documentation suite. Documentation costs have been known to run as high as 30% to 50% of military software.

Although one might speculate on many causes and effects, the fact remains that the people involved in the demonstration found themselves producing working code quicker than previously experienced. Most participants also discovered that in the course of 3 years, it was not an overwhelming task to understand another co-worker's code. This has to be attributed to the language's natural language reserve words and a concentrated effort to use meaningful labels for data objects as well as regular peer group reviews. Thus the maintainability issue, while not answered entirely in this project, certainly raised the expectation level that maintainable code could be produced using the Ada language. It is felt that the tools are available for producing maintainable code if correctly applied. These results appear in harmony with those obtained in the F-15 Eagle Dual-Control Augmentation System [8] where the microprocessors were programmed in Ada.

Some useful heuristics about programming in Ada were gleaned from the demonstration. These are listed below without justification or explanation:

1. Write simple code. Complexity occurs naturally in a military weapon.
2. Program first in Ada, regressing into another language only when necessary.
3. Design with maintenance in mind.
4. Use a comment block at the beginning of each program unit [9].
5. Create no Data label over one line in length.
6. Use "with" alone rather than "with" and "use".
7. When Ada code runs too slow, isolate inefficient Ada constructs and replace them if possible. Revert to another language as a last resort.
8. Do not expect Ada to be a panacea but a step in the right direction at least until the technology base matures.
9. Use the Language Reference Manual but supplement it with good books on Ada.
10. Package the constants and data types.

The last result was the software metrics developed. To measure such qualities as readability, maintainability, and reliability of Ada code, one requires an automatic tool. The Naval Postgraduate School in Monterey, California and the Naval Weapons Center entered into an agreement to develop a software metric program using graduate students working on master degrees in computer science. The first two students, Cmdr. Jeffrey Neider and Lt. Karl Fairbanks, completed the foundation program which performed static metrics on Ada source code [10]. A system has been developed that implements Halstead's length metric, Henry and Kafura's information flow, number of comment lines, readability assessment of the comments, and depth of the nesting level. This metric tool will continue to grow as more students contribute to the product.

It was rewarding to learn upon applying the static metrics to the IED code that the analysis indicated that the code was well above the acceptable levels. That the two groups worked fairly much independently of one another tends to lend credence to the results. The addition of more metrics into the base program and perhaps expanding to include some dynamic metrics are future plans.

In ι clusion, the answer to the rhetorical question posed in the title of this paper appears to be a qualified yes.

REFERENCES

1. R. J. A. Buhr. *System Design With Ada.* Inglewood Cliff, N.J., Prentice-Hall, 1984.

2. Edward Yourdon. *Techniques of Program Structure and Design.* New York, Yourdon, 1975.

3. Foreman and Goodenough. Carnegie-Mellon University, Software Engineering Institute, Pittsburg, Pa. *Ada Adoption Handbook: A Program Manager's Guide*, May 1987. Report CMU/SEI-87-TR-9.

4. Edward Yourdon. *Structured Walkthroughs.* New York, Yourdon, 1978.

5. T. Mossberg. Naval Weapons Center, China Lake, Calif. *Evaluating Ada Compilers for Embedded Systems*, 1987. NWC memo 3900-3922/146.3.

6. M. J. Haselhorst. Naval Weapons Center, Code 3922, China Lake, Calif. *Independent Exploratory Development Ada Demonstration Project*, 18 December 1987.

7. C. W. Hall. Naval Weapons Center, Code 3922, China Lake, Calif. *IED Ada Demonstration Project Software Development Plan*, 22 January 1988.

8. Westermeier and Hansen. "Ada in Use; A Digital Flight Control System," *AGARD Lecture Series*, AGARD-LS-146, 1986, pp. 10-1 through 10-9.

9. Nissen and Wallis. *Portability and Style in Ada.* London, Cambridge University Press, 1984.

10. J. Nieder and K. Fairbanks. Naval Postgraduate School, Monterey, Calif. *ADAMEASURE; An Ada Software Metric* (Thesis), March 1987.

Accelerated Convergence for Synchronous Approximate Agreement

J.P. Kearns*
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
USA

S.K. Park*
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
USA

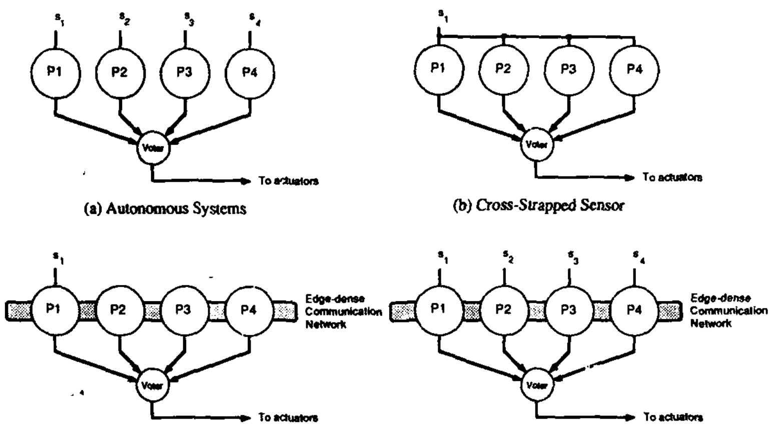
J.A. Sjogren
USAAVRADA/AVSCOM
NASA Langley Research Center
Hampton, VA 23665
USA

Abstract

The protocol for synchronous approximate agreement presented by Dolev *et al.*[7] exhibits the undesirable property that a faulty processor, by the dissemination of a value arbitrarily far removed from the values held by good processors, may delay the termination of the protocol by an arbitrary amount of time. Such behavior is clearly undesirable in a fault tolerant dynamic system subject to hard real-time constraints. This work presents a mechanism by which editing data suspected of being from Byzantine-failed processors can lead to quicker, predictable, convergence to an agreement value. Under specific assumptions about the nature of values transmitted by failed processors relative to those transmitted by good processors, we present Monte Carlo simulations whose quantitative results illustrate the trade-off between accelerated convergence and the accuracy of the value agreed upon.

1 Introduction

One approach to the construction of a fault tolerant computing system is through redundancy in the system hardware and/or software. Several alternative architectures for a redundant avionics subsystem, resilient to some number of hardware and software failures, are shown in Figure 1. In each alternative, four processors, P_1 through P_4, execute individual replicates of a particular software function. That function causes each processor to map digital sensor input into an output which is intended to drive some physical actuator or display. In alternatives (a) and (d), each processor has access to its own sensor. In alternatives (b) and (c), the processors share access to a single, presumably very reliable, sensor. The output of each



(a) Autonomous Systems
(b) Cross-Strapped Sensor
(c) Agreement on Single Sensor
(d) All Sensor Values Disseminated

Figure 1: A Redundant Subsystem

processor is a vote in an election to determine the sole output of the (sub)system. The redundancy in this subsystem is intended to tolerate the failure of any single processor or function (software). In the case of alternatives (a) and (d), the potential for resilience to sensor failure also exists.

The assumption that the function replicates fail *independently* (an important assumption for the efficacy of N-version programming[3,5]) has been empirically shown to be invalid in practice[11]. Also, analytical work suggests that the possibility of coincident errors in the function replicates may require a higher degree of redundancy than would be intuitively expected and that, in some situations, the reliability of the N-version system would be less than that of a 1-version system[8].

*Supported in part by the Army Avionics Research & Development Activity through NASA Grant NAG-1-787.

These problems, however, are irrelevant to the *structure* of a modularly redundant system. Their impact is felt in the degree of fault resilience, and it has been argued that careful design of the replicates and their integration into the system can lead to an acceptable level of resilience[1].

A key assumption in the preceding argument that redundancy provides resilience is that of *input congruence*. If the software functions are truly replicates of one another, then they are intended to compute the same output *if they are presented with the same set of inputs*. In the system of Figure 1, several intuitive alternatives for providing function input are available:

(a) Each function uses the data presented by its attached sensor.

(b) A single sensor provides input for the system. That sensor's output is cross-strapped to the input of all function replicates.

(c) A single sensor value is *reliably* broadcast to all replicates.

(d) The values of all sensors are *reliably* disseminated to all processors. Each processor then chooses some representative data value (presumably derived from the ensemble of sensor readings) as input for its function replicate.

The first alternative (a) is clearly not conducive to input congruence in that there is no coordination of sensors—each sensor delivers a value independent of the other sensors, and those values may be arbitrarily far apart. The second alternative (b) appears to gain input congruence at the expense of providing less fault tolerance—the chosen sensor becomes a critical single point of failure. The cross-strapping approach, however, generally offers no advantage for input congruence—wires break and output drivers go out of tolerance (an intuitively appealing presentation of this scenario may be found in [12]). The last two alternatives, (c) and (d), are, in fact, valid means of assuring input congruence. Their validity relies upon the term "reliable" as it applies to the dissemination of data in a network of processors connected by some communications medium. Of course, approach (c) is still vulnerable to the failure of the single sensor.

1.1 Reliable Dissemination Through Agreement

In a context characterized by faults of a totally unpredictable nature, the mechanism by which the reliable dissemination of data is achieved is through an *agreement protocol*. In the terminology of such protocols, a *General* (the transmitting process) disseminates a value to all *Lieutenants* (the receiving processes). The receivers must then engage in several *rounds* of message-exchange. At the end of those rounds the receivers may infer the value transmitted, if, indeed, the transmitter is non-faulty. Otherwise, the receivers may agree upon a null value which indicates a faulty transmission. More specifically, agreement requires that two conditions be met:

Agreement 1: All non-faulty recipient processes agree on the same value.

Agreement 2: If the transmitting process is non-faulty, all non-faulty recipients agree on the value transmitted.

In the absence of faults, exact agreement is trivially achieved by the transmitter's broadcast of the same data value to all receiving processes. In the presence of faults, the possibility of faulty behavior by any process, or possible corruption of any data message by the communications network, transforms a trivially solved problem into one which is decidedly non-trivial, both conceptually and in terms of the resources required to achieve a solution. Bit-by-bit agreement, with no assumptions about failure modes, may be achieved by a *Byzantine* agreement protocol [13]. Such protocols require substantial connectivity of the communications network (to avoid the forwarding of data through faulty processors), many rounds of message-passing (to assure that faulty processors cannot contribute to the agreement value), or message authentication through cryptographic means or through error-detection coding techniques (to limit the degree of faulty behavior). We note several salient facts about exact Byzantine agreement:

1. The system must be *synchronous*[10]. Intuitively, a synchronous protocol permits the determination of whether or not a remote process has crashed or is simply slow.

2. A system of n processes (not utilizing authenticated messages) can be made resilient to the failure of t processes, provided that $t < \frac{n}{3}$ [13].

3. At least $t+1$ rounds of message passing are required for exact Byzantine agreement under the most general assumptions[9] The original Byzantine agreement algorithm[13] required a total of $O(n^{t+1})$ messages; the best algorithm[6], in the worst case, requires $O(nt + t^3)$ messages.

A scan of these facts leads one to the conclusion that a Byzantine agreement protocol is typically extremely expensive in terms of replicated system components and communication bandwidth.

In the context of the system architectures of Figure 1, agreement would be utilized to disseminate sensor values in cases (c) and (d). In case (c), however, even with the reliable data dissemination provided by an agreement protocol, the entire system is vulnerable to inaccuracy of the sole sensor—erroneous sensor output could be reliably broadcast to all processors, resulting in unanimous consent on an output which is erroneous. In contrast, consider case (d) in which each processor could serve as the transmitter in a Byzantine agreement protocol (conceptually, one would envision four protocols, each with a different process as transmitter, concurrently in operation). At the conclusion of these (expensive) protocols, each processor would know the values of all four sensors, and could apply some (application specific) selection or averaging algorithm to choose a single "sensor value" as input to its function replicate. The validity of this scheme rests upon the facts that all non-faulty processors must have the same set of sensor data values as a consequence of the agreement-based dissemination and that the selection or averaging algorithm is the same at all processors.

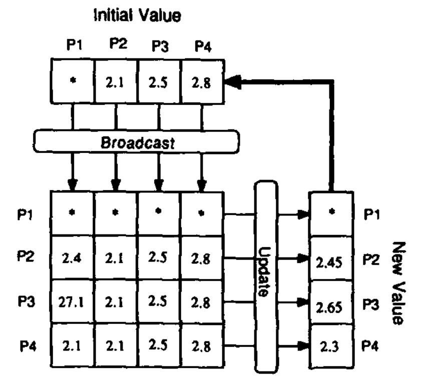
1.2 Approximate Agreement

An alternative approach to providing data to the function replicates of Figure 1(d) is through *approximate* agreement. That is, we require that all non-faulty processors derive an input value which must be "close to" the value derived by any other non-faulty processor, and that value must be within the range of the set of non-faulty sensor values. Informally speaking, these requirements are met by the approximate agreement protocol developed by Dolev *et al.*[7]. That protocol is intended to serve a network of n processors, at most $t < \frac{n}{3}$ of which may be faulty.[1] The maximum difference between any two final agreement values must be no larger than some *tolerance*, ϵ, a fundamental parameter of the protocol. The approximate agreement protocol operates as shown in Figure 2. This example depicts a four processor system ($n = 4$ and $t = 1$) in which the first processor is assumed to be faulty. During each step of the protocol. each processor maintains its current estimate of its agreement value; this is represented by the horizontal array at the top of the figure. The first action of a step of the protocol is each processor's broadcast of its value to all processors (including itself). In Figure 2 (a), the (i, j) entry of the matrix in the center represents the value received by P_i from P_j at the end of this round of message passing. P_1, the faulty processor, is shown sending different values to different processors, and, as illustrated, its multiset of received values is irrelevant—a faulty processor will not, in general, adhere to the approximate agreement protocol. Each processor then applies the same value update algorithm to its multiset of received values (i.e., its row of the matrix) as shown in Figure 2(b). For the case of a synchronous system, that update procedure involves sorting the multiset, discarding t values on each end of the range of the multiset, retaining the first and every t-th successive elements thereafter, and taking the arithmetic mean of the values so retained. It can be shown that a single application of the update function at all good processors will effectively reduce the *diameter*[2] of the multiset of values held by good processors by a factor of

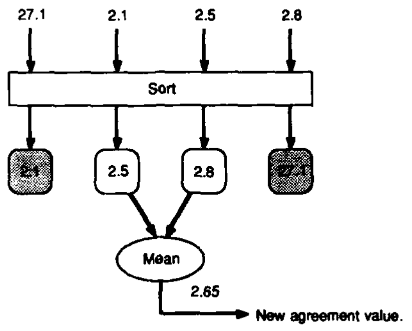
$$d = \left\lfloor \frac{n - 2t - 1}{t} \right\rfloor + 1.$$

Since $n > 3t$, this factor must be at least 2.

We note several fundamental differences between approximate agreement and (exact) Byzantine agreement: In Byzantine agreement, the agreement value is identical, on a bit-by-bit basis, for all non-faulty processors. In approximate agreement, the agreement value derived by any non-faulty processor must be within ϵ of the agreement value of any other non-faulty processor. We feel confident that the approximate agreement is all that is required in many specific applications (and, in fact, it may be *desirable* to avoid bit-by-bit agreement in an N-version programming system[2]). Moreover, (exact) Byzantine agreement is not even possible in an asynchronous system, but approximate agreement is easily implemented without global synchronization.



(a) One Round of the Protocol



(b) The Update Procedure

Figure 2: The Approximation Protocol ($n = 4$)

1.3 Halting the Approximate Agreement Protocol

Processors which are participating in a synchronous approximate agreement protocol each determine a halting round independently. This follows from the fact that a non-faulty processor, P_i, starting with an initial multiset of values V_i, can only pessimistically assume that the diameter of its multiset will be decreased by a factor of d on each round. Thus,

[1] Thus if $t = 1$, there must be at least three additional "good" processors, and if $t = 2$, there must be at least five additional non-faulty processors, etc.

[2] The *diameter* of a multiset of real numbers is the absolute value of the difference between the largest and smallest elements of that multiset.

processor P_i anticipates executing

$$H_i^1 = \left\lceil \log_d \left(\frac{\text{diameter}(V_i)}{\epsilon} \right) \right\rceil$$

rounds of the protocol in order to ensure that the multiset of values held by any good processor has a diameter no larger than ϵ. Two additional points concerning this halting technique are relevant: First, on the round after which a non-faulty processor halts, it broadcasts a **halt tag** with its agreement value on the next round of the protocol; after receipt of a tagged value from a halted processor, another processor will continue to use that tagged value as the value transmitted (without receipt of an actual message). Secondly, a processor whose value is not received within a timeout interval (presumably representative of the maximum message transmission time), will be assumed to have send some arbitrary default value.

The fundamental difficulty with this halting technique defined by H_i^1 is that H_i^1 can be arbitrarily large; i.e., *faulty processors, by the initial broadcast of values which are arbitrarily far removed from actual values, can arbitrarily enlarge the diameter of the initial multisets of good processors, and consequently, arbitrarily delay convergence.*

2 Accelerated Convergence

In this section we will develop and empirically evaluate two approaches to faster convergence: *adaptive halting* and *initial multiset editing*. The degree to which each approach succeeds depends on knowledge of the expected behavior of the system in which the protocol operates. We adopt the following assumptions about the system:

Initialization: Each good processor independently draws an initial value (i.e., will read its sensor) from a distribution of "good values". For the purposes of our simulations, we define this distribution to be $N(0,1)$, the standard normal distribution.

Broadcast: Whenever a *faulty* processor transmits a message, it sends a value drawn independently from a distribution of "bad values". We utilize $N(0, \alpha) : \alpha \geq 1.0$ as the family of bad value distributions.

We make no distinction between the failure of a processor and the failure of a sensor; the two constitute a single entity as far as fault behavior is concerned. In the following discussion, when we refer to a "failed processor", we actually refer to the processor/sensor pair. All faults are assumed to be independent of previous fault behavior and the current state of the system. Our model, therefore, does not address the notions of correlated faults or of processors failing in coordinated fashion. We term this model the *two-faced independent fault model*, and we utilize this model in the remainder of this paper.

Note that values transmitted by faulty processors share the same mean with the values chosen by good processors. This facet of the model was intended to represent worst-case behavior—faulty values chosen from a distribution whose mean is substantially different from the mean of the distribution of good values are easily discarded by the approximate agreement update algorithm.

2.1 Adaptive Halting

The *adaptive halting* procedure is outlined below:

- Let V_i^r be the multiset of (good and bad) values held by processor P_i at the conclusion of the r-th round of the protocol. Let H_i^r be P_i's computed halt round at the end of the r-th round of the protocol. H_i^1 is the previously defined static halt round for P_i as defined in the original protocol.
- After accumulating V_i^{r+1} in the broadcast phase of round $r+1$, compute

$$\text{temp}_i = \max \left(0, \left\lceil \log_d \left(\frac{\text{diameter}(V_i^{r+1})}{\epsilon} \right) \right\rceil \right).$$

- Set H_i^{r+1} to $\min(H_i^r, r + 1 + \text{temp}_i)$.

The obvious goal of this approach is to exploit the round-by-round variation in V_i^r to achieve much faster convergence than that guaranteed by the pessimistic convergence factor d. For example, suppose that a single faulty processor (P_1), in a 4-processor system, $\{P_1, P_2, P_3, P_4\}$, consistently broadcasts a common value which is significantly different from the valid sensor values on the first round. The original protocol will then compute the same large halt rounds for all good processors based on their common inflated initial multisets. Moreover, the agreement values computed by the good processors on the first round will agree *exactly*, and thus all rounds after the first will be unnecessary overhead. If P_1, at some later round, consistently broadcasts a common value relatively close to the agreement values, the the adaptive halting procedure will then provide quicker termination. Adaptive halting can be shown not to invalidate any of the functional correctness properties of the original approximate agreement protocol.

Figure 3 presents the results of adaptive halting applied to four processor and seven processor systems. As in all experiments presented in this paper, $\epsilon = 0.1$. Larger values of ϵ (all other system parameters remaining constant) merely shift the plots downward, and smaller values shift the plots upward. One of the processors in the four processor system, and two of the processors in the seven processor system, exhibit two-faced independent failure. The vertical axis of the plots is the mean halting round for a Monte Carlo simulation (1000 replications) of the protocol in operation for a specific value of α. The halting round on any replication is defined to be the largest halt round for all non-faulty processors.

The results are disappointing but intuitiuve. Adaptive halting provides negligible gains over the traditional pessimistic approach. In effect, the probability that a faulty process broadcasts a commmon "nearly good" value is so small that the

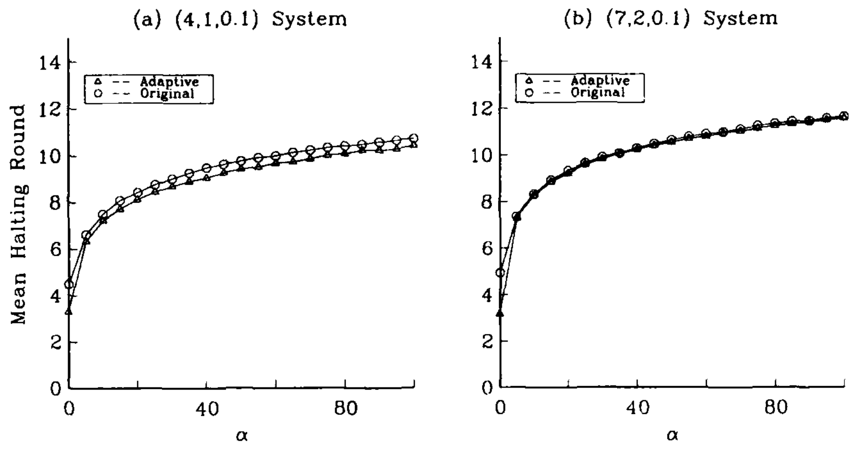


Figure 3: Adaptive Halting

adaptive halting round only rarely becomes smaller than $H_i^!$ *for all non-faulty processors*. The trend towards less gain in larger systems is also intuitive: adaptive halting succeeds only when bad values are somewhat consistently disseminated to good processors, and the greater the number of processors the more difficult it is to lie consistently under the two-faced independent failure model. We would expect that adaptive halting would be valuable only in a failure regime characterized by processors which operate correctly most of the time but which fail wildly for only brief intervals. Thus we reject adaptive halting as a general approach for accelerated convergence and turn our attention to the possibility of editing sensor data to achieve faster convergence without introducing error into the agreement protocol.

2.2 Editing the Initial Multiset

If we consider the application of approximate agreement in a real-time fault-tolerant *control* system, we gain some leverage by which the halting difficulty may be overcome. Figure 4 illustrates the basic form of the control system software. We

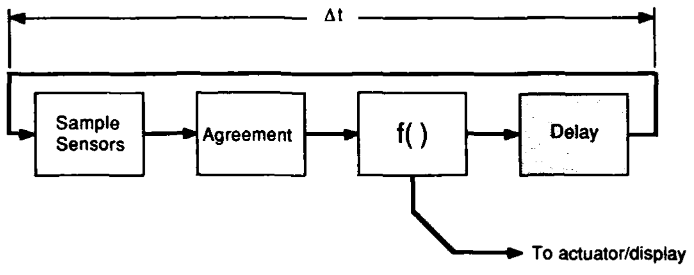


Figure 4: Control Loop

assume that the control loop is frequently executed—a 30Hz loop would not be uncommon. A consequence of this frequency is that the sensor readings will presumably not change "too much" on successive iterations. For example, if the sensor were an aircraft accelerometer, the dynamics of the aircraft and the integrity of the airframe itself would impose bounds on the rate at which acceleration can change. Therefore, we assume that certain application-specific criteria can be used to rationally edit bad data values from the initial multisets accumulated in the first round of the approximate agreement protocol.

In order to verify that data editing can provide substantial performance benefits, we consider two theoretical bounds on the best possible halting behavior of an approximate agreement protocol.

Absolute Optimal The earliest possible halting would be achieved by globally monitoring, on each round, the agreement values computed by all good processors. When the diameter of that multiset becomes less than ϵ, halt. Of course, there is no practical way to discriminate between faulty and non-faulty processors or to globally monitor agreement values (although global snapshots[4,14] could be used, at great expense, to monitor the values).

Practical Optimal If we compute halting rounds based on the initial multisets of values from only non-faulty processors, we achieve the best possible halting performance for the original approach to halting. Note that we still cannot discriminate absolutely between good values and bad, and thus, this bound is also not directly implementable.

Figure 5 demonstrates that there is an appreciable gap between the stopping behavior of the original protocol (the upper line in both cases) and both idealized techniques. Note specifically that the separation between the original algorithm's performance and that of the practical optimal technique is five or six rounds of message passing (and updating) for relatively large values of α (in both systems). That gap translates into the potential for substantial real-time savings via editing the initial multiset. If we could omnisciently edit, i.e., remove only faulty processors' values from the initial multisets, the halting behavior would be exactly that of the practical optimal bound. The separation between the practical optimal bound and the absolute optimal indicates the price we pay for not being able to identify failed processors as the algorithm executes. For both four and seven processor systems, that gap is roughly 1.5 rounds.

The *initial multiset editing* approach is based on wild-point editing as outlined below:

- Let μ_G be the mean of the distribution of good sensor values, and let σ_G be its standard deviation. Let $V_i = \{v_1, \ldots, v_n\}$ be the multiset of values obtained by processor P_i in the broadcast phase of the initial round of the approximate agreement protocol.
- For all $v_j \in V_i$, if $|v_j - \mu_G| > \beta\sigma_G$, then replace v_j with μ_G.
- Calculate H_i^1 based on the edited initial multiset.

In an operational system, historical data could be used to provide statistical estimates of μ_G and σ_G, and the threshold parameter β would be chosen according to the needs of the application. There is an important trade-off here. The larger the value of β, the less likely valid data is to be discarded and the slower the rate of convergence. Conversely, smaller β implies faster convergence at the possible expense of an error. Here an error actually means non-convergence; i.e., either the diameter of the final agreement values of non-faulty processors is greater than ϵ or an agreement value falls outside of the initial range of sensor values read by non-faulty processors.

Figure 6 shows the result of initial multiset editing applied to four processor and seven processor systems. The threshold parameter, β, is varied between 1.0 and 3.0.

Results of Figure 6 must be evaluated in the light of the possibility of errors which may be produced by the rejection of valid data. Unlike the ineffective adaptive halting procedure, data editing may result in non-convergence—clearly, the

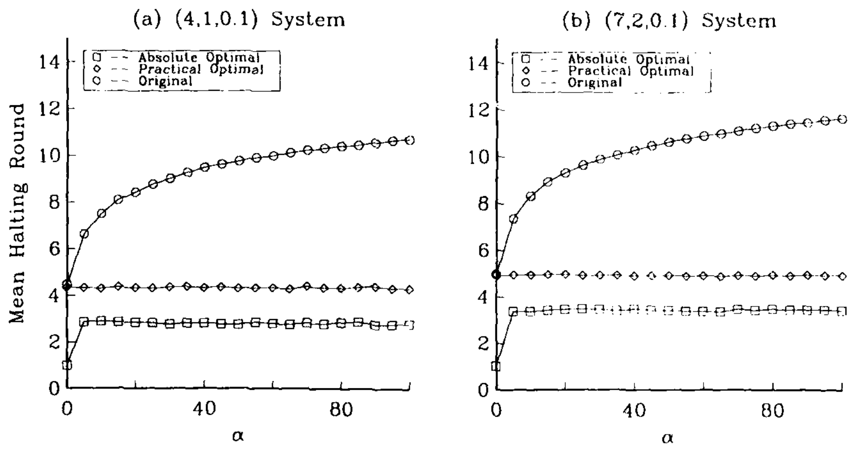


Figure 5: Bounds on Performance of Editing

possibility of discarding data from a good sensor/processor perverts any formal correctness argument developed for standard approximate agreement. Table 1 shows the worst-case performance for a given choice of β in both systems. Worst-case performance here is simply the highest percentage of errors for 1000 replications for any α.

We make the following two inferences from Figure 6 and Table 1:

- Halting performance which is significantly better than practical optimal ($\beta \leq 1.0$) can be achieved—but only at the expense of a non-negligible error rate.
- With a reasonably large rejection threshold parameter (β = 2 or 3), halting performance asymptotically equal to the practical optimal is achieved under the two-faced independent failure model. An experiment in which β = 2.0 and

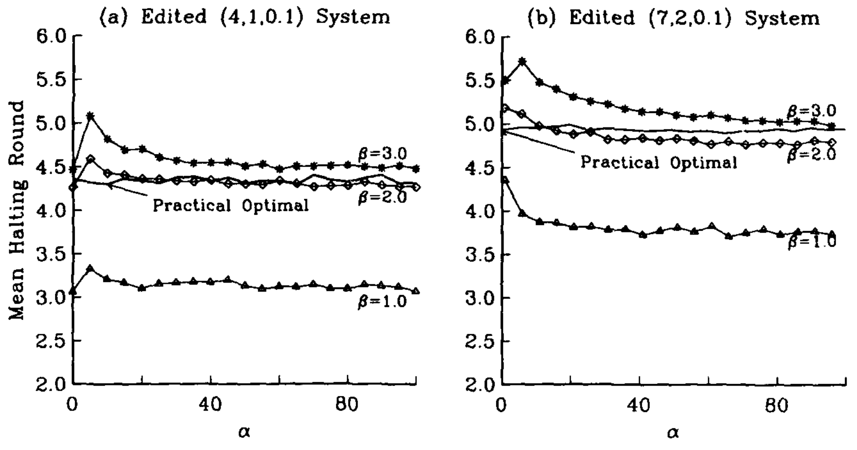


Figure 6: Halting with Edited Initial Multisets

$\alpha = 50.0$ executed for over three million replications with only two errors.

3 Concluding Remarks

We have presented a model of fault behavior under which data editing can yield substantial performance benefits without introducing appreciable error into the approximate agreement protocol. Under our model, a rejection threshold of two or three standard deviations from the mean of the distribution of good values leads to near optimal performance without introducing appreciable error into the system. We postulate that historical data, accumulated during the operation of a control loop, may be used as a basis for this editing. Clearly, the editing tends to damp the control system somewhat, but a legitimate change in sensor value will be accommodated immediately.

System	4-1	7-2
β	Error Rate	Error Rate
0.5	27.2	11.2
1.0	4.8	0.6
1.5	0.7	0.1
2.0	0.1	0.0
2.5	0.0	0.0
3.0	0.0	0.0

Table 1: Errors Induced By Editing

We are in the process of the analytic modeling of the editing procedure. This work will allow validation of the simulation results and permit rapid extrapolation into new system contexts. The issue of how one chooses ϵ and β is clearly application specific. We are currently beginning to examine real sensor (accelerometer) data to better understand how that choice impacts upon the end-to-end performance of the control system (e.g., if the loop iteration frequency is sufficiently high, perhaps an occasional agreement error would have no appreciable effect on the performance of the system). The issue of the degree of damping induced by the data editing will also be examined in the light of real sensor data.

We postulate that approximate agreement can be a useful construct in the design and implementation of real-time reliable systems. Further, adapting the basic protocol in response to knowledge about expected input values and the demands of the application is likely to be essential in order to meet performance specifications.

References

[1] L.S. Alger and G.L. Greeley. *A Highly-Reliable Architecture for Executing N-Version Software*. Technical Report CSDL-P-2727, The Charles Stark Draper Laboratory, Inc., Cambridge, MA, 1986. (Presented at 18th Joint Services Data Exchange for Inertial Systems, 1986).

[2] P.E. Ammann and J.C. Knight. Data diversity: an approach to software fault tolerance. In *Digest of Papers 17th International Symposium on Fault-Tolerant Computing*, pages 122–126, 1987.

[3] A. Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-

11(12):1491–1501, 1985.

[4] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] L. Chen and A. Avižienis. N-version programming: a fault-tolerance approach to the design of reliable software. In *Digest of Papers 8th International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978.

[6] D. Dolev, M. Fischer, R. Fowler, N. Lynch, and R. Strong. Efficient Byzantine agreement without authentication. *Information and Control*, 53(3):257–274, 1982.

[7] D. Dolev, N.A. Lynch, S.S. Pinter, E.W. Stark, and W.E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.

[8] D.E. Eckhardt, Jr. and L.D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, 1985.

[9] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[10] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, 1985.

[11] J.C. Knight and N.G. Leveson. An experimental investigation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, 1986.

[12] G.M. Papadopoulos. Redundancy management of synchronous and asynchronous systems. In *AGARD Lecture Series No. 143*, pages 7-1–7-12, 1985.

[13] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[14] M. Spezialetti and J.P. Kearns. Efficient distributed snapshots. In *Proceedings 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

DISCUSSION

R.Busser, US

What is your view on the "N-version software as a technique for tolerance to generic software errors" controversy?

Author's Reply

My view is to use it at your own risk. Knight-Leveson provided evidence that coincident error phenomena can occur in the "moderate" reliability range (version reliability ~ 10^{-4}, 10^{-5}). But even extrapolating successful experiments in this range to an "ultra" level (10^{-8}) cannot be justified. Hardware failure independence relies on accepted physical principles, but will never be "proved" statistically due to the number of tests required. Software development has no such known principles.

ROBUST ALGORITHM SYNCHRONIZES MODE CHANGES IN FAULT-TOLERANT ASYNCHRONOUS ARCHITECTURES

Ing. Antonio Silva
AGUSTA SISTEMI S.p.A.
Via Isonzo 33 21049 TRADATE ITALY

ABSTRACT:

The need for protection against common-mode failures, in Hardware and in Software, as well as the need to cut Hardware costs and complexity, has led to redundant asynchronous architectures.
The resulting random-phase sampled-data systems are loosely coupled, reducing fault propagation,but need to make homogeneous decisions whenever certain event combinations occur.
Since decisions are basically boolean operations, a method has been developed to achieve homogeneous agreement on discrete signals,without imposing constraints on input or architectural characteristics to the system designer.
The method solves the critical problem of synchronized change of operational mode in avionic systems, where a safe "all or none" way of making decisions is of primary importance.

INTRODUCTION

Modern technology for Guidance, Weapon and Supervisory Systems in Aircraft and Missiles is nowadays trying to meet the ever-growing demand for performance, together with extensive Fault-tolerance and operational safety characteristics.

Distributed computing, organizing the raw processing power of a number of microprocessors in a synergetic structure, provides a way to achieve considerable performances, allowing to implement multiple redundancies, as well as *cross-monitoring* and related fault management policies, to achieve an overall high degree of Fault-tolerance.

The use of multiple redundancy has brought forward the major issues of Common Mode Failures, capable to defeat the System.
The availability of several microprocessor families with comparable computing power gives now the opportunity to incorporate dissimilarity in the redundancy scheme, for instance, allowing different processors to monitor each other.

Common Mode Failures not covered by dissimilarity are those caused by particular input data values, yielding wrong results, due to flaws in the Hardware or in Software. Since any computing system is basically a sampled-data system, this may happen in all nodes if all nodes are sampling data exactly at the same instant, i.e. if the system is synchronous.

To overcome this, Asynchronous Distributed Architectures have been developed, where the nodes are deliberately out of phase by a random percentage of the basic cycle rate (frame) that remains however the same for all nodes (isochronysm) to keep the system to a manageable complexity.
The chances that the flaw shows up in all nodes are reduced and , as a side-effect benefit, the System Design is forced towards loosely-coupled Architectures, effective in fault-propagation prevention.
Another major advantage of such architectures is the absence of critical common Hardware, in charge of providing all nodes with synchronizing signals.

The use of distributed Architectures brings in problems, the most obvious of which, and certainly not the least significant is that the processors, or , more generally, the computing nodes, will have to communicate with each other to permit both synergy and cross-checking.
It is actually essential for the System to ensure that different nodes do not generate contrasting commands or messages or take decisions in opposite directions.

SYNCHRONISATION (or Equalisation): the Analogue Case

It is relatively easy to synchronize (or consolidate , or equalize) a single analogue datum starting from signals coming from multiple asynchronous sources.
Several techniques, such as Median Selection, plain or weighted Averaging, Distance Compensation with Filtering, Dynamic Regression or combinations of the above (up to the complexity of a Kalman Filter) have been

developed and are widely used to calculate in each asynchronous node a single datum that is sufficiently similar to that calculated in any other node.

These techniques can be easily made (or are intrinsically) insensitive to the failure of any single node , incorporating faulty node datum isolation, and in general guarantee that the output datum , for each node, is constantly drifting, more or less smoothly, towards a datum that is continuously calculated , according to the technique used, as an appropriate "average" of all valid input signals; this implies they are intrinsically robust and safe.

Furthermore, in multiple sampled data systems, the difference in the sampled values in any node pair cannot exceed an upper bound, rigorously determined by the signal's dynamics and the worst-case delay in sampling from one node to another. In general, though, this upper bound is significant when the system is sampling data with a basic cycle that is so slow to be comparable with Shannon's Theorem's minimum rate; in practical systems a basic cycle at least faster than that at least by an order of magnitude is no more than sensible practice.

SYNCHRONIZATION (or Consolidation): The Discrete Case

Treating signals with boolean values obviously prevents the use of techniques such as averaging or filters. Actually, considering the signals in continuous time, the techniques used for analogue signals are capable to converge, if the signals show a steady state of sufficient duration, but the algorithm output, with a final threshold to obtain again a boolean, yields unacceptable transient differences, causing a subset of nodes to consolidate a boolean opposite to the other nodes.

Another difficulty is that, treating booleans coming from asynchronous sources in the boolean domain, no upper bound can be established that limits the difference in output from one node to the others. The instantaneous difference in the output from any two nodes can be either 0% or 100%, and the only type of error upper limit that can be determined is in terms of phase and duration of the output inequality.

Inequality can be caused in fact by various mechanisms:

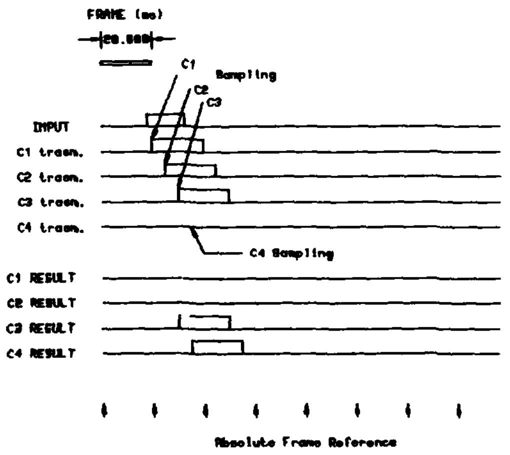
- The sampling process applied to discretes causes the following:
  - Pulses shorter than a full basic cycle can remain undetected in a subset of the nodes.
  - Every sampled level is stretched in duration to an integer number of basic cycles (frames).

- Different phase in sampling causes the following:
  - Opposite levels sampling on transition boundary
  - Internode communications appear with additional delay at each node
  - Any input pulse may be measured with a 1-frame duration difference in any node pair.
  - Voting and threshold mechanisms of any kind fail to consolidate consistently in all nodes.

A typical example showing voting techniques inadequacy for this case follows:

Consider four nodes, C1..C4 , with a basic frame cycle τ , where C1 is synchronized with an absolute frame of period τ, while C2,C3,C4 are out of phase by 1,2 and 3 times $\tau/4$, respectively. Let us assume that the algorithm is defined as:

If 3 out of 4 among the available signal sources agree, the output is the agreed level, otherwise it remains unchanged from the previous frame.
The 4 sources are the direct input and the signals read by the other three nodes, and it is assumed that the time from sampling the input and transmission to the other nodes is short with respect to the frame.



It becomes evident that if a pulse of duration $1/2\tau<\Delta<3/4\tau$ starts just before the start of the absolute frame, it will be directly sampled by C1,C2 and C3, but not by C4. But C1..C3 transmit their sample to each other and to C4 before C4 samples its inputs. The result is that C1 and C2 will not satisfy the algorithm condition for agreement, while C3 and C4 will, even if C4, for instance, takes its decision in contrast with its own sampled value .

Threshold techniques fail as well, since they are basically voting on the event of a certain time being elapsed with certain conditions, and time measurements can differ by one frame, causing the threshold to be exceeded only in a subset of the nodes in the asynchronous case.

SYNCHRONIZATION OF MODES: The nature of the Problem

The major problem in multinodal systems is that of synchronization of operational mode changes in consequence of events of discrete nature: external switches, reaching of thresholds, timeouts ,etc. In synchronous systems , typically equipped with a "data input interrupt", all nodes are sampling data at the same instant, and thus differences may only be due to malfunctions. Transmission of samples from one node to others is synchronized as well, and the transmission delay is a known constant, in absolute terms. Appropriate algorithms guarantee discrete signals consolidation in a minimum number of frames, simultaneously in all nodes.

In Asynchronous multinodal systems this holds no longer. The problem changes from the selection of an algorithm that consolidates the output in the minimum number of frames to the use of an algorithm that allows consolidation at all.
In most such systems, discrete events are in fact the key data on which decisions are made about the operational mode the whole system should assume. Examples are Manoeuvre Mode, Attitude Hold, Trim, Trim Synchronization, Automatic transitions, and so on. Note how a mode change of this sort causes in most cases a re-initialization of long-term datums in the system's Control Laws, and from this point of view a transition has long memory, or "latches"; in other cases a true latching process is effectively required.
This means that, for each node consolidating a transition, the following processing is altered.

This makes a Synchronized Consolidation of the mode transition compulsory, putting time response at a lower priority. The actual need is in fact to ensure that the whole system changes its operational mode in the same way, or does not change mode at all. In other words, the priority is to achieve system's consistency at all times.

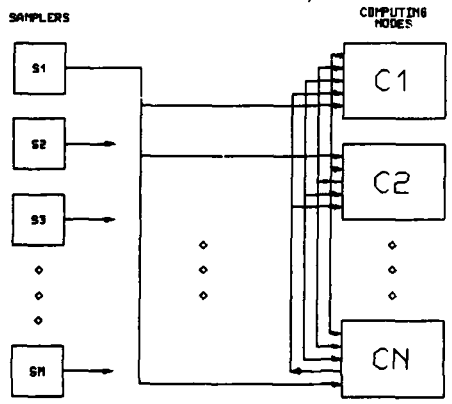
Systems with multiple cross-checking redundancies are particularly critical in this respect, since even temporary inconsistencies leading to different operational modes in different nodes can cause the checks to detect failures (that actually do not exist) that will generate cut-out commands. This occurs for certain when the decision result is latched in a subset of the nodes.

To overcome this, the only alternative to a proper synchronizing algorithm is to decrease the cross-checking sensitivity, reducing error detection time response and redundancy management performances, that is not acceptable in most cases.

A TYPICAL ASYNCHRONOUS ARCHITECTURE

Let us now consider a typical architecture with multiple asynchronous sampling and computing nodes.

Consider N computing nodes, C1..CN, able to communicate to each other and receiving data from the outside world via M sampling nodes, S1..SM, able to provide fresh data to C1..CN with a frequency higher than the basic frame τ of the System (equal in period for all {C} nodes).
Let us assume that {S} refresh the data to {C} in "broadcast" mode, i.e. that data is available to any node in {C} at the same instant. Communications between the nodes are assumed "broadcast" as well, and each node is able to transmit data also to itself. Internodal communication must be such to guarantee complete transfer of relevant data from Ci to Cj in less than a frame time τ.



Each Ci will freeze, at the beginning of each of its frames, and for the entire frame duration, the whole set of input data, consisting of :

- data from {S}
- data from the other nodes
- data from itself

and the "staleness" of any of these data is $<= \tau$.

Note how, from the point of view of the computing nodes {C}, the whole set of data coming from S1..SH can be considered simply a set of external asynchronous data, and as such the presence of the samplers {S} will be ignored in the following.

THE SYNCHRONIZATION PROBLEM IN DETAIL

Since each Ci is a sampler with sampling rate τ, the phase relation between Ci and Cj is in fact more of interest than the phase relation of Si to {C}. It is always possible, without loss of generality, to rename the members of {C} so that C1 is the node receiving first the new value of a certain piece of data, and all other nodes C2..CN sample the same data in sequence, with CN sampling with a delay Δ with respect to C1, where $0<=\Delta<\tau$

Note that, on the contrary, it is important to consider the timing of internodal communication, since these repeat with a period τ and thus have an effect at the sampling rate. The isochronism of such communication and the sampling in all nodes , even if the transmission bursts are out of phase, is the key to a simple solution to the synchronization problem.

The worst case of asynchronism may, in virtue of the possibility to renumber the nodes, be simulated by an even temporal distribution of nodes over the period τ, with the first node synchronous with an absolute sampler of period τ, without losing generality, since sampling makes a phase delay of τ/N identical to any other phase delay in the range $[0..\tau)$.

The case of two or more nodes in synchronism is simpler than the case above, and is anyway covered by the proposed method .

Let now *t0* be the generic absolute time of a transition in a discrete input, externally generated or produced by the Software (exceeding a threshold, a timeout, or combinations of these with other discretes).
Due to sampling, each node Ci will read the input at absolute time $t_i>=t0$, but , for the same reason, if esists at least one node out of phase with respect to C1, there exists an infinite number of transitions , with appropriate phase and duration ranges, such that for a subset of {C} t_i does not exist, i.e. a subset of {C} does not recognize the transition. This is true for transitions shorter than τ, and can be generalized to the leading or trailing edge of any discrete pulse, if the total duration of the pulse is not equal to an integer number of frames.
This means that any measurement of an external event's duration is subject to a one-frame possible difference in any node pair. Note, however, that the maximum difference in measurement between any two nodes cannot exceed one frame, since any part of a pulse longer than one frame is certainly recognizeable by all nodes.

Nevertheless, we can conclude that, in an asynchronous system, even if , in a continuous absolute time reference the patterns of a discrete input are similar, in the discrete reference system tied to a single node, the patterns transmitted by all nodes may show considerable difference.

The analysis of the problem with multiple trials in several directions and bearing in mind the constraints imposed by the real-time nature of the systems, has generated the following requirements for a mode synchronization algorithm:

- independent operation: the algorithm must be the same in all nodes and must be executed independently at the same point in each node's basic cycle τ.

- insensitivity to noise: the algorithm must guarantee a *synchronized response in the whole system* , even in the presence of fast and random transients in the inputs.

- pseudo-synchronization on output: all the nodes must synchronize the transition with a maximum difference between each other shorter than τ.

- robustness: it must be impossible for any one node to ignore a transition synchronized by other nodes.

- reconfigurability: the algorithm must be easily extendable to include features like error detection and faulty node masking.

- efficiency: the communications load between nodes, being the system strictly connected, must be reduced to the minimum. Additionally, since the algorithm must be executed every frame in all nodes, it must be computationally efficient, and must allow a substantial parallelism in inputs and outputs.

- time response: the basic algorithm must have the minimum time response to a signal going to a steady state condition that ensure the requirements above to be met. It must, however, be easily extendable to filter out transients shorter than an arbitrary duration.

DESCRIPTION OF THE ALGORITHM

The proposed algorithm is based on a most intuitive consideration:

If the System is composed of asynchronous samplers and none of the nodes knows its phase relationship with the others, the single node is not able to conclude about an agreement among all nodes, in any selected instant .

If however we consider all the discrete signals coming from all nodes in the system as continuous signals in the time reference of a single node, and a time window is examined with a sufficient duration (integer number of frames) in the immediate past, every node is able to compare the pattern of a signal, for all nodes, as it has evolved in time.

The comparison between all the temporal images of the same signal as seen by all the nodes allows to define a concept of "retrospective agreement", that means: if an agreement cannot be sure for the current sample, it may be possible in the past, if the patterns in the past are close enough for a sufficient duration.

The algorithm is more precisely defined as follows:

Given that:

1 All nodes are sampling all inputs and execute the algorithm every frame of duration τ

2 Each node communicates to all nodes in the system (including itself) the value of the direct inputs as sampled by it within a time $\ll\tau$ from the sampling instant.

3 Every node applies the algorithm to the images coming from the other nodes and from itself (the latter read during the previous frame (see note)).

Then the result is calculated as follows:

* Chosen a window of N frames, if the patterns received from *ALL* nodes show a common level of the signal in at least N-1 consecutive frames, then the result is made equal to the common level.

* If none of the two logic levels satisfies this condition, the logic value of the result remains equal to the one at the previous frame.

NOTE : This last clause is not mandatory , since a difference of a full frame is inessential, but is useful to make the following discussion simpler.

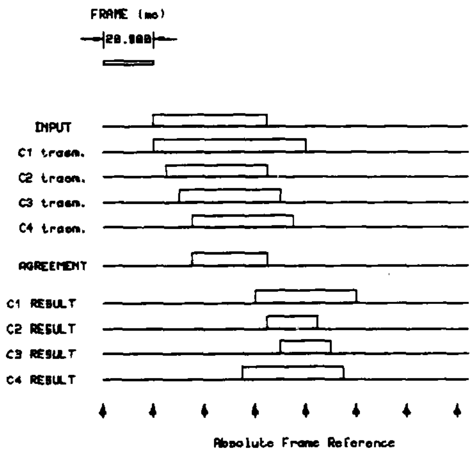
From the clauses 1 and 2 we can say that:

a) the maximum duration difference for any of the transmitted signals, as sampled by a node reading the communication data, is limited to one frame, source to source, since the isochronism of the samplers limits the differences only to phase effects.

b) for the same reasons, the maximum difference, in terms of temporal position of the patterns, is one frame.

The algorithm will then certainly consider agreed all signals with a steady state duration > (N-1)* τ, and will behave as an absolute filter for all signals with steady state duration < (N-2)* τ.
For signals with a steady state duration within the limits above,the result depends on the phase relation between the nodes, but yields the same result in all nodes, independently, with a difference in the result phase and duration between any two nodes limited to one frame.
The full agreement required by the algorithm automatically limits the pattern steady-state duration to the minimum recognized in any one node.
Relative to the signals from the other sources, this minimum signal is either in phase or out of phase by one frame, and anyway always in the same sense, since otherwise the worst case phase shift between any two nodes would be 2 frames, contradicting b).

The fact that the patterns have to be recognized within a time window that is one frame longer than the pattern target length accommodates for both duration and phase differences , and guarantees that the event of recognizing the agreement happens (with a tolerance of one frame in absolute time) in all nodes.
The differences between results among the nodes in terms of resulting pulse duration and phase can be easily explained in an absolute time reference.
The algorithm has in fact been conceived in absolute terms, since the signal relations are more evident and remain undistorted by the sampling in a particular node.
Let us consider a simple example consisting of 4 completely asynchronous nodes, C1..C4 , equally distributed , in phase, on the frame τ. (Note that this is also a worst case for synchronization).
The signals retransmitted by each node after sampling are shown. The effect of sampling is evident, stretching the transmitted signals to integer multiples of τ. The effect of phase relations is also evident, both on sampled values and on the phase of retransmission.



In continuous terms, it appears clear how the conditions a) and b) are satisfied, but interesting is the virtual signal that identifies , in the continuous time, the full agreement between all four nodes. This signal is the temporal intersection of the signals transmitted by all nodes, and as such is unique (in the continuous time) and is shorter than or equal to any of the transmitted signals.

If this signal comprises N-1 contiguous sampling instants for any node, then ALL nodes will recognize an N-1 frames pattern in all signals in an N-frame window.
In fact the full agreement signal starts synchronously with the sample of the latest node , C4 , and , to contain N-1 samples for it, must be of duration greater than (N-2)* τ .
But, since this signal is the temporal intersection of all transmitted signals, this means that *ALL* these signals have a duration greater than (N-2)*τ and , since the signals come from sampling, their duration cannot be but an integer multiple of τ, and so must be at least (N-1)*τ.

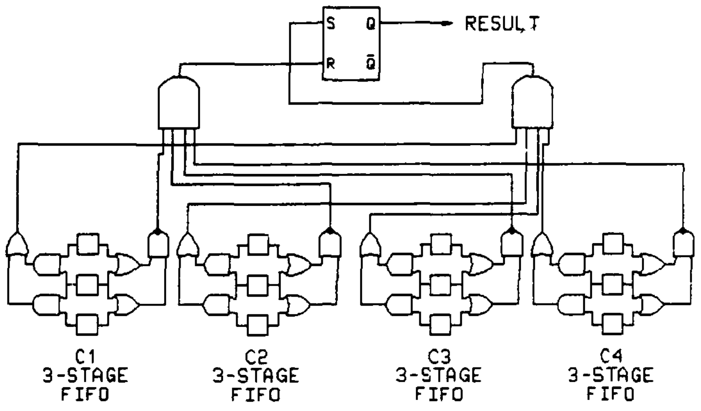
This implies that at least N-1 contiguous samples are present in all the patterns, and , since the phase relation is limited within one frame, the patterns are recognized within an N-frame window in *ALL* nodes.

On the contrary, if no node exists for which the full agreement signal comprises N-1 samples, this means that at least for one node the transmitted signal is shorter than N-1 frames, i.e, N-2 frames or less, and as such no node will be able to recognize an N-1 frame pattern in all signals within the window.

This technique can be applied to both logic levels independently, and for N > 3 the mutual exclusion of agreement on the two levels is intrinsically guaranteed, and so no ambiguities are possible.

When no agreement can be reached on either level, i.e. during fast transitions of the input, or in response to short pulses, the algorithm does not change the result from that obtained at the previous frame. This decision may be considered questionable, but, also in the light of examples, and considering that in a sampled data system short pulses may be lost due to its nature anyway, seems the most sensible approach; obviously, the result will have to be properly initialized at startup, and the signals history as well, to obtain a consistent steady state.

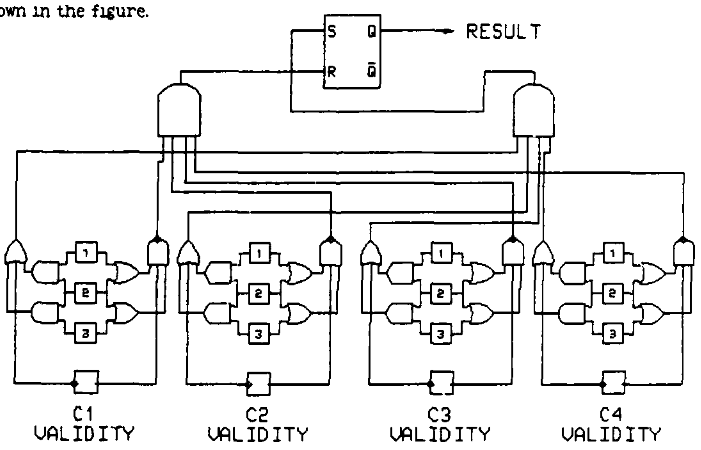
The following logic network depicts this algorithm for N = 3 , working on both logic levels, with 4 asynchronous nodes.



Note that result retention in case of no agreement has been implemented in the simplest way, tying the two agreement signals to the inputs of a S/R Flip Flop. Since the two agreement signals are mutually exclusive and the lack of agreement does not alter the Flip-Flop status, its output Q is the desired result.

FAULT MASKING EXTENSION

In a redundant system, it is common practice to implement complex Built-in Test functions, capable to isolate malfunctioning nodes. The algorithm can be extended to incorporate a "masking" of faulty nodes' data, simply forcing the pattern recognition signals for such nodes to their active state, before the final full agreement gate, as shown in the figure.



The agreement is then dependent only on the signals coming from the working nodes, since the agreement of faulty nodes is forced unconditionally.

ERROR DETECTION AND ISOLATION CAPABILITIES

This algorithm can be extended to include two forms of error detection and management: faulty node data isolation and failsafe operation.

- Faulty node data isolation.

Apparently, this algorithm , although safe, can lead to a stuck system only because a single failed node remains in disagreement with the others.
This condition, however, if the actual input discretes are not in continued transition, can be synchronized as any other boolean among all nodes, or at least among all working nodes.
If the failure of a node is such to deeply alter its functions, the Built-in Test functions will isolate that node within reasonable time. If instead the failure is more subtle, like reading a single discrete in a stuck level in that node, the agreement among all nodes (possibly excluding the suspect node), on the fact that that node has been in disagreement with all others for a sufficient period of time, guarantees that the set of working nodes synchronously decides to exclude it, and, if the suspect node is able to run the algorithm, it is able to decide, independently and synchronously, to shut itself down.

- Failsafe operation.

In the case when common mode errors affect all nodes of one type, and no majority agreement is reachable, it is however always possible to reach an agreement on an unresolved situation timed out, and on this basis to force the result to a failsafe level in all working nodes. This condition must latch and appropriate messages must be sent to the outside world.

EFFICIENCY IN CASE OF MULTIPLE DISCRETE INPUTS

Since the algorithm is based only on logical operations on bits stored in memory, and since the majority of processing units is able to operate simultaneously on all the bits in an arithmetic unit's register, if the history of signals belonging to the same class, i.e. with same N-frame window, is kept in memory with all bits "packed" in words , it is possible to apply the algorithm on the entire word, obtaining a word result and reducing execution time by a factor equal to the number of discretes processed in parallel.
The masking operations may be done in parallel , as well as fault isolation and reversion to failsafe result.

CONCLUSIONS

Asynchronous redundant architectures have long been regarded as risky, in terms of possible artifacts occurring due to their nature and difficult to anticipate, in particular in management of boolean entities. This algorithm is believed to relieve much of these concerns and to allow such architectures to fully exploit their capabilities, with a reasonable price in terms of computational loading.
Asynchronous loosely coupled architectures, including simpler and cheaper hardware configuration, no hardware weak points, independent processing, lower chance of common mode failures and overall benefits in terms of cost and reliability , can be employed without the deterrent of extensive communication needs to solve synchronization issues.

Definitions and Requirements for Distributed Real-Time Systems

by

Christina Berggren
International Business Machines Corporation
P.O. Box 6
Endicott, NY 13760 USA

ABSTRACT

This paper defines the performance parameters for distributed real-time control systems and describes the differences in requirements between these and the other major category of distributed real-time systems: resource sharing. Examples of explicit values of the performance parameters are given for some applications. Finally, an evaluation is made of the applicability of existing and emerging communications standards to real-time distributed control systems.

CATEGORIES OF REAL-TIME SYSTEMS

There are two major categories of real-time distributed systems: control and resource sharing systems. The major difference is that control systems have hard deadlines to meet and resource sharing systems do not. A hard deadline means that a late result is equal to system failure. The communication requirements for these two types of applications are vastly different.

Resource sharing systems are general purpose information processing applications sharing distributed services and resources. These shared services can be physical resources such as data bases, processors, and printers, or they can be shared information possibly, but not necessarily, generated in real-time such as electronic mail, voice, telemetry, or graphics. Resource sharing applications may involve large amounts of data, and sometimes they are referred to as Information or Data Processing Systems.

These systems are 'open'; that is, they are not predefined, and they include many different types of resources. In addition to reliability, the main requirements driver is commonality between heterogeneous resources. Real-time for these applications means 'real fast' and can, as a rule, be measured in terms of the level of 'human inconvenience'; there are no firm deadlines to meet, and there is no severe penalty for an occasional long response.

Resource sharing systems are not further addressed in this paper; the applications are well understood and the technology exists. Their communication requirements can be met by the standards intended and developed for these applications, the OSI (Open System Interconnect) Reference Model and IEEE-802.X or the emerging ANSI FDDI, for example.

Control systems differ in communication requirements significantly from those described above, and the technologies required are nonexistent or at the 'cutting edge' stage. Industrial control applications are usually referred to as process control systems.

These real-time process and control systems are not 'open'; their applications are specific and their resources dedicated to the application. They are tightly coupled in time and can be characterized by the fact that they have firm deadlines to meet, and failure to meet the deadline is critical and possibly life threatening. Their applications are often so large or complex that they require more than one computational resource concurrently: the distribution of the task itself.

These control systems are complete entities with defined allocation of resources and predictable communications. They are usually of a limited geographical distribution such as a military platform or a chemical processing plant, but this may not necessarily be true. SDI, for instance, will fall in this category.

Control systems do not require connectivity to a large number of undefined different manufacturers' equipment; two-three to a dozen may be enough for any one application. The connectivity can be supported by one (or a couple) physical medium and a limited number of protocols. Of course, these applications do require interoperability between similar equipment from different vendors for cost effectiveness.

Distribution of a task over several computational resources also requires the distribution of execution control to said resources. The requirements of this distribution of execution control cannot be satisfied with the statistical quality criteria of the communication standards mentioned above. Deterministic low latency quality control is required for time distribution, task synchronization, and sensor/effector data integrity.

CONTROL SYSTEMS PERFORMANCE PARAMETERS

Real-time for control systems can be quantified. Performance parameters include the control loop, resolution, and the distribution of control, where the distribution of control can further be broken down into latency, periodic traffic jitter, order, and accuracy. Degree of fault tolerance required and automation are system level parameters which impact the design and performance of the communications.

CONTROL LOOP

The process control term for control loop is fundamental frequency. It is a measure of the degree of real-time. It is the total time allowed for a specific task measured from the input of a sensor to the completion of the reaction at an affector. Most control systems have more than one control loop: hierarchies of control loops.

Examples of explicit requirements for the control loop for some different applications are as follows.

In the recent Stark incident in the Persian Gulf, the control loop was 2 minutes. That was the time the Stark had to intercept and destroy the missile starting from the sensors detecting the release.

For the Space Shuttle Mission Operational System (MOS), the ground-based mission manager, the control loop is 1/2 second.

For avionic simulators, the control loop is 90 or 150 ms, depending on whether the aircraft is a tactical fighter or a helicopter with nap-of-the-earth capability or a transport aircraft. This delay is the interval between the pilot (in the simulated cockpit) changing aircraft controls and seeing an indication of the change on the control panel.

For oil refineries, there are a few 1 second control loops, but most loops are between 8 and 64 seconds.

The New York city Traffic Control system has local 1 second loops changing green/red light allocation at major intersections as measured starting from the sensor in the street. The loop to change traffic patterns for all of the city is 10 minutes.

A juggling robot being designed at IBM's Research division in Yorktown has control loops of 100 ms and 300 ms. The 300 ms loop is from throw to catch of the puck by the mechanical hands. The 100 ms loop is from the decision to move the receiving hand to the actual catch.

For tactical fighters, the control loop (one of many) is 10 ms.

RESOLUTION

The process control term for resolution is processing frequency. It is the frequency of repetition of either computation, sensor sampling, or actuator/effector reactions. It is a measure of system granularity, and most control systems are hybrids with various resolutions in different functional areas of the system. For instance, sensor sampling is often at a higher resolution than the main computational resolution with filtering and preprocessing of the sensor data.

Resolution is sometimes measured in time but most often in Hz (or cycles per unit of time). Examples of explicit resolution requirements for some control system applications are as follows.

The Shuttle Mission Operational System has resolutions of 10, 2, and 1 Hz in different functional areas of the system.

The New York city traffic control system samples sensors in the street at 60 Hz, while the update rate for the total system is every 10 minutes. Local traffic light changes are more frequent. The 60 Hz resolution may seem high, but it allows the speed of the cars to be measured, thus directly indicating the system operational effectiveness.

The resolution of the juggling robot is 60 Hz.

Avionic simulators at present operate at 5, 20, and 60 Hz. Some are hybrids with aircraft sensor data updated at 20 Hz and the out-the-window displays at 60 Hz. The sensor data resolution requirements are gated by the resolution of the aircraft system being simulated and those of the out-the-window displays by human perception.

For oil refineries, the processing frequency is 3 Hz but may involve the multiplexing of several hundred data points per second. The resolution is limited by the recovery time of the flying capacitor analog input multiplexor.

The resolution requirement for tactical fighters in development is 500 Hz. Advanced radar applications presently have a resolution of 250 Hz and IFF (Identification Friend or Foe), 200 Hz. There also are classified applications with resolution requirements of 50 kHz.

CONTROL DISTRIBUTION: LATENCY

Latency here is defined as the task-to-task delay, which includes the communication protocols. It may be a function of the number of stages in the processing pipeline and is dependent on the system design. For data-driven applications, it is from the completion of one task to the initiation of the next.

Latency is measured in time and varies with the type of communication service required such as acknowledged, connection oriented, etc.. Examples of latency requirements for some applications are as follows.

An avionic simulation with 90 ms control loop can be implemented with a task-to-task delay of 5 ms. The juggling robot requires 1 to 2 ms latency and the classified application 4 μs. A system designed to control a nuclear power plant in Belgium has 1000 data points, all of which have to be delivered in less than 10 ms, and the tactical fighter has delay requirements for certain messages of 1 ms.

CONTROL DISTRIBUTION: PERIODIC TRAFFIC JITTER

The process control term for periodic traffic jitter is scan frequency variability. It defines the degree of precision for time distribution, task synchronization, and sensor/effector data integrity. Allowable jitter in periodic traffic as a rule is a function of the resolution; the higher the resolution the smaller the allowable jitter.

Periodic traffic jitter is measured in time. Examples of periodic traffic jitter for some applications are as follows.

For the 60 Hz avionic simulation, a periodic traffic jitter of 0.5 ms is acceptable. For advanced radar applications, 50 μs is acceptable. For the French tactical fighter, 200 μs jitter is deemed acceptable, while the British fighter design requirement is 50 μs. The classified application needs periodic traffic jitter not to exceed 15 to 20 μs. The Architectural Control Document for the U.S. Space Station calls out a requirement for time distribution to an accuracy of 10 μs.

CONTROL DISTRIBUTION: ORDER

For process control systems, the sister term is sequence of events or SOE. For instance, during process upsets, order can be extremely important for many petrochemical processes. During the process upset, many alarm conditions may occur, and it is necessary to know their order to correct the cause rather than one of the symptoms.

For data-driven distributed applications, guaranteed message order of arrival is essential for some message categories. Other distributed applications require guaranteed message order of arrival during degraded system states for anomaly handling when making best-effort decisions based on partial data sets.

CONTROL DISTRIBUTION: ACCURACY

Accuracy is the differential reception time. It is the delta in time between reception at the first and last destinations of a message transmitted simultaneously to more than one destination. For a physical broadcast medium, it is the medium propagation delay between destinations. For token passing rings, it is a function of the station delay and the medium propagation delay.

FAULT-TOLERANCE

Fault-tolerance is a major requirements driver for control systems. Most control applications require uninterrupted operational performance with extremely short or no 'hiccup'; that is, instantaneous fault detection, isolation, and reconfiguration. Other control applications require a graceful degradation after a fault with continued operation in degraded mode.

The containment of the failures is also essential for process control systems; they must not ripple through the system. The process control term is scale of failure.

The petrochemical industry has been at the forefront of control technology because hydrocarbon processes tend to run away and self-destruct. The primary concern has been for safety for the equipment and human life. Production rates and product consistency and quality have been of secondary importance. This is reflected in the computer programming for these systems; about 10% of the effort (lines of code) is concerned with actual control functions and the remaining 90% with control integrity.

For military applications, availability and performance are of paramount importance. From Chernobyl we have recent knowledge of what scale of failure can mean for a process control system, but it is difficult to envision the consequences of a 'runaway' or incorrectly operating military system. Therefore, the requirements of these distributed real-time control applications also include a high degree of fault-tolerance for equipment failures. This impacts all layers of the communications protocol, and it is a major difference from resource sharing systems.

APPLICABILITY OF COMMUNICATION STANDARDS TO CONTROL SYSTEMS

LOWER LAYER PROTOCOLS

After an independent evaluation board unanimously agreed that only a token passing ring could meet all the requirements in the HART Document (1), the Avionic Systems Committee of the Society of Automotive Engineers (SAE-AS2) examined existing and emerging token passing ring technology for applicability to distributed control systems.

The IEEE 802.5 could not meet the throughput requirements, and increasing the clock rate was not a viable solution. FDDI, an emerging ANSI standard, did meet the throughput requirements, but with a severe bandwidth penalty if access performance is required. Simulation data has shown, for instance, that for a guaranteed required access of 1 ms, which does not meet the explicit requirements documented in this paper, only 20% of the FDDI bandwidth could be utilized (2).

It was also determined that the FDDI access method was ill-suited for the distribution of control. Its priority mechanism cannot guarantee the order of arrival; higher priority messages do not necessarily get transmitted before those of lower priority. And for stations with messages pending of the same priority, access is not a fair round robin but dependent on the physical location of the station (3).

A detailed comparison of the two protocols (FDDI and HSRB) was performed for space applications. This study included simulation data of the two in the scenario of the benchmark tests specified in the HART document. The results showed that the two performed about equally when all messages were of the same priority, but the FDDI timers could not be set to support the priority separation.

After the SAE committee established that neither the FDDI nor the IEEE 802.5 protocol could be modified to meet the requirements set forth in the HART document, the committee set out to develop a token passing ring standard specifically intended to address distributed fault-tolerant real-time control applications. The High Speed Ring Bus (HSRB) was released in April of 1987 as Draft SAE Standard AS4074.2 (3). Many Department of Defense standards are generated by this international body.

The HSRB media access protocol supports low, fully deterministic message latency through message based priority reservation. Its priority operation can guarantee message order of arrival. Enforcement of the priority operation on all messages is a system design option that may be dynamically selected at a small bandwidth penalty. An optional Multiple Short Message protocol bypassing the priority operation allows the insertion of up to 16 messages shorter than the ring latency before priority again is enforced.

Periodic traffic jitter for the HSRB can be controlled by priority assignment, and it is a function of the maximum allowed message length in the system design. For instance, if highest priority is allocated to the time distribution task, different levels of precision can be achieved by controlling the message length, as illustrated in Table 1.

Table 1. HSRB Periodic Traffic Jitter

Max Msg Length	Max Synchronization Jitter	
Number of Bytes	Priority Order Selected (ms)	Priority Order Not Selected (ms)
2048	0.40	0.45
1024	0.20	0.35
512	0.10	0.30
32	0.01	0.25

The second column is when priority order is not selected and multiple short messages are allowed. Jitter for a second periodic message assigned next highest priority will add roughly 50% to those values.

To meet the fault-tolerance requirements of the intended applications, the HSRB protocol with dual counterrotating rings supports high speed automatic physical reconfiguration. This means stations can be inserted or removed or a broken fiber bypassed in less than a millisecond. This is done in hardware and does not require any software negotiation.

HSRB supports two addressing modes: physical and logical. The logical mode provides for the addressing of tasks rather than hardware stations. This way, a task (or its station) communicating with another task need not be aware of what processor it is transmitting to, thus eliminating the need for routing directories and some of the time consuming processing traditionally done by communication protocols.

Logical addressing also provides for software containment of failures. If a processor goes down, for instance, only the processor replacing it is affected. Other processors containing tasks that were communicating with tasks in the failed processor need not be notified through loading of new configuration tables since the physical locations of other tasks are transparent. This way, the HSRB protocol allows for the containment of failures.

Automation was also a requirements driver for the HSRB logical addressing. Artificial Intelligence tasks for the HSRB intended applications require rapid parallel searches of distributed dynamically updated data bases. Logical addressing on the physical medium provides this capability.

UPPER LAYER PROTOCOLS

Standards do not yet exist for the upper layers of the communications protocol required to support distributed real-time fault-tolerant control applications.

OSI, the only widely accepted standard for upper layer protocols, is much too slow. For instance, performance measurements taken at the National Bureau of Standards' Network Laboratory indicate best-case, end-to-end delays of 33 ms (5), excluding the uppermost three layers.

Faced with the immediate need for low overhead protocols several years ago, IBM's Federal Systems Division embarked on the development of a next-generation communications architecture for its Advanced Systems Development Laboratory (ASDL).

Performance measurements on the lowest layer of the three layer ASDL architecture using Proteon's 80 Mb/s token passing ring indicate the feasibility of achieving the end-to-end delays required for real-time control systems.

In addition to being too slow, the OSI Reference Model can not be adapted to distribute control in the form of time or task synchronization. Many required services do not presently exist, even though some additions are being worked into the model, and the OSI allocation of existing functions to specific layer is not optimized for real-time control applications.

Some of the OSI deficiencies include acknowledged datagrams, predefined connection-oriented service, nonacknowledged connection-oriented service, concentration capabilities, and automatic invocation. Required multicast/broadcast is presently not included in the OSI standard but it is being worked.

Another capability required for the SAE HSRB intended applications that is lacking is refresh. Kalman filters, for instance, may need up to five sets of the most current periodic data points with automatic overwriting of the oldest data, other cyclic applications may need two or three data points with the oldest set refreshed at a periodic rate.

However, some of OSI developed methodology does apply to control systems. The philosophy and structure and many of the goals do; only the intended applications are different. Upper layer protocols under development for real-time control systems take advantage of OSI methodology, including the partitioning into layers of functional capability, peer entity associations, and the layer interface concepts of service primitives and service access points.

As with the OSI Reference Model, the Europeans have been on the forefront of this development, recognizing several years ago that there was a need to develop communication standards specifically for distributed control applications. A proposal on the applicability of the OSI model to shipboard systems was presented by an Italian delegate at a NATO meeting as early as 1973, and the French government is presently supporting the definition of a real-time model called the GAM-T-103.

Within the United States, the SAE-AS2 Communication System Requirements Subcommittee is the major standards forum for these applications. Driven primarily by the desire for interoperability of a strong combined defense, the SAE is also the focal point for NATO country technology exchange.

REFERENCES

1. HART Requirements for the SAE/AE-9B High Speed Data Bus, ISSUE 4, April 22, 1986
2. A. Goyal and D. Dias: Performance of Priority Protocols on High Speed Token Ring Networks, IEEE/ IFIP 3rd International Conference on Data Communication Systems and their Performance, June 1987.
3. D. Dykeman and W. Bux: An Investigation of the FDDI Media Access Control Protocol, IBM Research Report 1987
4. SAE Draft Standard AS4074.2 High Speed Ring Bus (HSRB), April 14, 1987
5. C. Franx and K. Mills: Open System Interconnection for Real-Time Factory Communications: Performance Results (National Bureau of Standards)

USE OF MARKOV PROBABILITY AND RELIABILITY MODEL GENERATION METHODS IN THE ANALYSIS OF RELIABILITY OF A FAULT TOLERANT, HARDWARE AND SOFTWARE BASED SYSTEM WITH FLEXIBLE REPAIR POLICIES

By Dr Paul White
Rolls-Royce Plc
PO Box 31
Derby
DE2 8BJ

SUMMARY

This paper begins by considering some of the problems that have arisen in the application of traditional reliability methods to fault-tolerant systems, particularly with the widely-used fault tree approach. The application of the Markov state-flow equation to reliability analysis is then considered and it is shown how many of the aforementioned problems disappear with this approach and how the basic equation can be manipulated to include repair policies, discrete events, and to calculate system reliability. Then the issues are taken on to the next step by considering how to set up a reliability model from system design information in such a way as to ensure the Markov states and transitions are correct and so as ensure that the reliability analysis gives an upper bound for system failure. The concept of formulation of design information and automatic generation of a reliability model for any given system is explained and an example analysis given based on a typical jet engine control system.

1 INTRODUCTION

Traditionally reliability predictions for industrial systems are widely done using the fault tree method of approach. Henley and Kumamoto (reference 1) trace the development of fault trees from their hazy origins out of the previously erroneous "strong as the weakest member" philosophy. Fault trees have been widely taken on board throughout industrial engineering as the standard method of representing and calculating system reliability, to the extent that the method is regarded as the natural one in the minds of most engineers, and any tendencies of the method towards a residue of errors and omissions and inaccuracies become either glossed over or attacked on an "ad hoc" basis. However, when a system is being analysed which substantially breaches the limitations of the traditional methods so that the end product becomes one in which the errors, omissions, and inaccuracies become potentially serious and the final reliability prediction cannot be justified to any great extent or only with great difficulty then a new approach is required. The Markov Probability Analysis (MPA) method has its roots in an ancient technique but is gaining credibility as an alternative method for system reliability prediction. The Reliability Model Generation (RMG) method goes further and overcomes the problems not addressed by the Markov method.

2 FAULT TREES AND PROBLEMS

Henley and Kumamoto (reference 1) show that the credibility of the "strong as the weakest link" philosophy suffered in the German rocket programme of the 1940's and the idea began to emerge that the system reliability could be better estimated by considering the effects of random failures in all the components. For many years after that the systems being analysed consisted largely of non-redundant non-fault-tolerant designs. The fault trees for such systems consisted largely of a collection of base events feeding into a single 'OR' gate at the top, the reason for such simplicity being inherent in the nature of the available hardware of the time, and the capability of carrying redundant parts in any given system. Such redundancy and back-up systems as did exist tended to be fairly simple and usually built-on afterwards. Whilst it is stretching the point a bit to say this, it was almost a case of designing a system without redundancy and then if a back-up was needed it would be by means of another complete system. Reliability predictions for such systems could be obtained by considering the rate of occurrence of random failures in each event that feeds into the top 'OR' gate and doing the arithmetic accordingly.

As time went by and technology moved on, particularly with the advent into common use of electronics of first analogue and then digital type, systems could be designed with redundancy so that very much smaller, and in time not-so-expensive, parts could be backed-up in a way that enables many faults to be tolerated within the system. Fault trees for such systems began to grow alarmingly and throw up new problems to be overcome in order to obtain the reliability prediction with any degree of confidence.

The first such problem concerns "exposure" and "snap-shot" times. System reliability is traditionally expressed in terms of rate of occurrence per hour or in some other similar units. For non-redundant systems when the overall reliability is largely made up of single random events then the calculation is relatively straightforward. However, when combinations of events are considered then the calculations are based upon probabilities of the combined events occurring together or in sequence. In order to carry out such calculations the traditional method is to assign to each contributing random event an "exposure" time representing a typical period over which the system is exposed to that fault and dividing the top event probability by the appropriate "snap-shot" time inherent in the specified exposure times. Thus for fault-tolerant systems there is always some kind of time factor appearing in the system reliability calculation. The end product is a fault tree and corresponding calculation which assumes a snap-shot period within which the probability of system failure is considered. The question arises as to what snap-shot and exposure times should be used in the calculation? This question is particularly pertinent to dormant faults or faults which are not repaired except following another failure. The ideal snap-shot period to take is the time at which it is known that the system will be cleared of all existing faults. But it is often more meaningful to take the snap-shot period as covering a particular system "run" which then raises the issue of setting exposure times of faults that could already exist in the system before the "run" commences, in such a way that the snap-shot period could be considered typical.

The second problem is that of significance of faults. This issue affects the inclusion or exclusion of certain faults in the fault tree depending on the relative failure rates and exposure times, and also affects the order of accuracy of the reliability calculation. The reliability engineer often has to make a judgement as to what to include in a fault tree analysis and the temptation is there to go for a trade-off of calculation complexity versus calculation accuracy. But, particularly where dormant faults are concerned, this can lead to omissions that are potentially significant. In theory everything should be considered but in practice this is not so.

The third problem concerns fault ordering which addresses the issue of whether any particular fault has to occur before another in order to cause total system failure. This tends to occur quite frequently in systems where there are back-up possibilities, particularly where the detection (and hence accommodation) of a given fault could be affected by previously occurring faults. Consider a system with a fault tree represented by one top 'OR' gate and a lower 'AND' gate in one branch. If the required order of occurrence of faults in even such a simple system is altered this can cause non-simple changes to the system reliability calculation. When considering order of repair of faults also the debates that can arise concerning factors or other such considerations in reliability analysis are apparently limitless!

The fourth problem is that of fault-dependence which arises out of the possibility of including the same fault in different branches of the fault tree in a way that leads to errors in the calculation. It is difficult for all but simple systems to write down the fault tree exactly corresponding to the correct Boolean expression for failure combinations without the possibility of omitting something or including the same thing twice. For instance, if the same fault can occur in 2 branches leading to an 'AND' gate then although the Boolean logic may be correct the reliability calculation is in danger of assuming 2 faults must occur where only one actually happens.

The fifth problem is that of fault-propogation which describes the possibility that a fault in a system may cause other parts of the system to fail also. For instance, if a dual-redundant system is powered electrically from a single power supply then failure of that power supply would cause the failure of all the dual-redundant parts also and lead to system failure. These effects can often be simple, but also can often be complex and lead to errors or omissions in the analysis.

The sixth problem that can arise is the fundamental one of determining operational requirements of the system. This is conceptually the reverse of the system failure requirements i.e. a statement of that which is needed to maintain system operation. This is usually the way round that a system is designed and so is more easily linked to the system design assumptions. Also operational requirements tend to include an element of "preference" whereby the redundancy is not as simple as saying "either this or that" but instead defines which set of system parts are in the active control loop at any given moment depending on what faults have already occurred. This notion then requires that faults in the active control set (as it is defined at the time of the fault occurrence) will need to be treated more seriously in the analysis than faults in the currently redundant parts. Traditional reliability analysis methods do not have a ready made way of picking this up and can overlook its implications particularly where the active control set has a fair degree of flexibility.

The seventh problem is concerned with the various accommodation methods that can exist within a system so enabling it to reconfigure following the detection of a fault within it. This may include hardware accommodation whereby a like or similar part is used in place of the failed part, or software accommodation whereby a digital-based system will opt to perform its calculations based on the availability of an alternative set of hardware functions. This latter type of accommodation is increasing in availability and flexibility in current industrial systems.

The eighth problem concerns the detectability and detection of faults. This issue is linked in with the operational requirements in that an undetected fault needs to be treated more seriously than a detected fault. For a start, where an undetected fault can lie dormant in the system for a long period of time, then there is a steadily increasing possibility of a real system problem due to that undetected fault being included in the active control set. It goes without saying that a fault cannot be accommodated unless it is some way detected within the system and cannot be repaired unless it is some way known to the operator. Thus system reliability is clouded by the at-the-time definition of the active control set parts and the existence of and operation of a detection method and by the accommodation if detected.

The ninth such problem or consideration is that of repair and maintenance which addresses the issue of which faults can be put right in a faulty system and at which particular times. These issues do have an impact on the reliability of fault tolerant systems due to the effect on the levels of redundancy in the system at the start of a typical system run. This is known to affect the "exposure" and "snap-shot" times to be assumed in fault trees but in the present age with much more complex systems operating than hitherto the possibility arises that different fault combinations will be repaired at difference times and traditional analysis methods have great difficulty in reflecting these complex situations. Suppose, for instance, there are 10 different potential faults within a system then there are at least 1024 system configurations some of which may lead to total system failure. Even if the fault tree is itself relatively simple the nature of the faults and of the repair policy that exists to deal with them may not be and it can be seen that the analysis rapidly gets out of hand. Such a repair policy is quite possible if a digital-based system is used to flag up to the operator which faults exist within the system.

This brief scan illustrates the sort of considerations which beset traditional reliability methods, none of which apply to non-redundant systems. After all, there is no need to detect a fault within the system if the system cannot function following the occurrence of the fault anyway. Nor is there a need to consider fault-dependance or fault-propogation for single fault causes of total system failure, and the active control set of such systems is the whole system, and the accommodation methods will not exist. Repair has to be immediate, that is to say at the end of each system run. For todays fault-tolerant systems there is an abundance of opportunity for debate about the failure analysis and for errors, omissions, and inaccuracies to creep in.

There is thus a great need for and benefit from analysis techniques which seek to automate and regulate for all these issues. There is no shortage of fault tree packages which take the heat out of drawing and calculating system reliability in this way. But the one thing such packages insist upon is that the user specifies the reliability model and the failure rates and the exposure times and the above considerations show that it is there that errors, omissions, and assumptions can occur. The Markov probability technique obviates some of those problems by looking at the analysis on a single-fault as-it-happens basis and here too there is a growing number of analysis packages which also require that the reliability model is in place. The capacity of such packages to handle "non-standard" items such as repair and other discrete transitions is generally either limited or too complex to be readily usable. Thus there is seen to be a great need to automate the generation of the reliability model itself including repair considerations so that the model will be automatically produced from design assumptions that are as simple and meaningful as possible. This paper describes the mathematics and concepts of reliability model generation techniques and their automation with an introduction to how they have been implemented in a package developed at Rolls-Royce plc.

3 MARKOV PROBABILITY ANALYSIS (MPA)

The method proposed by Markov in 1907 is a mathematical concept which divides any given system into a number of "states", and by considering the "rate of transition" between the states builds up a simulation of the system.

This is easily adaptable to system reliability analysis. The system states in this case serve to categorise each configuration of faults within the system and the reconfigurations that may have taken place following those faults. Each state is different from each other state in terms of the fault combinations and system reconfigurations it describes although each state may contain various such combinations within itself. One of the states represents a fully-operative zero-deficient system and one other state represents all the various fault combinations that can cause the total system failure (sometimes referred to as the "death" state). The number of states in between these extremes is a function of the system complexity. The value each state may take on at any given time t is the probability that the system is in that state at that time t and so the sum of all the state values at any given time is always 1. Each non-zero element of the transition matrix represents a "flow" of probability out of one state into another. The Markov mathematics is given in Appendix A.

3.1 MARKOV CONCEPTS AND PROBLEM SOLUTIONS

The Markov method has some success in tackling the aforementioned problems of reliability analysis. One obvious advantage that immediately emerges from using this method is that basing the equations around a "flow" out of one state and into another ensures that the total system probability adds up to 1 at all times. Thus any inaccuracies in the fault tree method due to making approximations are eliminated and a parity-type check on probability exists in the Markov method that the fault tree or other methods do not have. Another obvious advantage that emerges is that the reliability analysis reduces to that of considering the effect of single fault transitions on the system. This obviates the need to formulate the failure cut-sets directly and thus reduces error potential.

The basic Markov state-flow differential equation may be solved analytically (if practical) or by an appropriate numerical method. Any problems caused by estimation of exposure times is thus eliminated, as the state probabilities are constantly changing as time proceeds. The need for a snap-shot period is largely dispensed with as such a simulation, particularly when executed by a computer process, can run for as long as is desired. The snap-shot period effect is therefore limited to the issue of how long to run such a simulation and not concerned with the need to choose the period and exposure times to be compatible with each other and to reflect a "typical" period.

The Markov method by the implication mentioned above of completeness of probability lends itself more naturally (than traditional methods) towards the inclusion of all fault combinations and letting the "significance" issue run its course through the model.

Fault ordering problems are largely overcome with the Markov technique. This is achieved by defining different states for cases where the same faults have occurred but in a different order and only combining them into one state if and when it is clear that the order didn't matter after all. The application of single faults to each given state is a much more sure way of determining the various paths to system failure than that of trying to write them down straight-off.

The issues of fault-dependence, fault-propogation, operational requirements, accommodation, and detectability are at the very core of the choice of states and transitions in the Markov model of the system. One approach to this is to resurrect the fault-tree mentality and try simply from certain system knowledge to define in some way what the system states should be and then fit the known system information into the various state transitions. It follows that errors and omissions made in the fault tree approach can also creep in by doing this. This method of defining a Markov model in some empirical fashion when used in reference 2 led to a case where 2 distinct states had been erroneously combined into 1. The error was shown up when applying the Reliability Model Generation method to that example, as that method conceptually does a painstaking "no assumptions" approach to setting up the states.

Repair and maintenance is an issue also more readily addressed using the Markov method than by more traditional methods. Each state in a Markov model of a given system represents one, or maybe more, fault combinations. Each level of repair provided for in the system operating procedures can be interpreted in terms of which states are covered by that level of repair. So at whatever time periods within the model simulation that each repair level is supposed to be carried out, the simulation interprets this in terms of a discrete probability change. The states which are known to consist of fault combinations which should be repaired at this level will have their state probabilities set to zero, and other states (usually the "everything working" state) will have their probabilities increased accordingly. After this process is done the next simulated run may proceed. The mathematics of this are contained in Appendix A.

In the literature various attempts exist to use the Markov method in the realm of reliability analysis. For instance Gai, Harrison, and Luppold (reference 3) have attempted to demonstrate the use of the method for the reliability analysis of a dual-redundant engine controller which is similar in nature to the example of section 5. The approach made in reference 3 was to define the states in a reasonable kind of "as we go along" approach with some reference to the hardware units within the system. This quickly became complex to the extent that it was necessary to make some assumptions to avoid the number of states from getting out of hand, a problem that would have been avoided by the automatic state generation and storage capability of the Reliability Model Generation (RMG) method. Also some fairly specific assumptions concerning system repair were made in this paper, whereas repair policies can become quite complex. In another publication Butler (reference 4) describes a computer package for reliability analysis based around the Markov probability approach using some mathematical results by White (reference 5) and Lee (reference 6) giving upper and lower bounds for system probabilities. This package has the flexibility to include non-random fault transitions and to consider system reconfigurations as time-dependent processes, but requires a lot of work to set up the states and transitions and associated numerical data and also does not permit consideration of discrete repair policies being carried out at specific times.

3.2 MATHEMATICS OF MARKOV IMPLEMENTATION

The following mathematical treatment is detailed in Appendix A. The basic Markov equation represent the fault and state transition effects on state probabilities during a particular "run" of the system i.e. when it cannot or will not be repaired. It is quite possible for the Q matrix to be variable. An aeroplane flight is a typical example of where the operational and stress environment of the system depends on whether take-off, cruise, approach, or taxing is currently going on. A time simulation can handle this failure rate variability in a numerical way with no problem at all. In this way the calculation of $\underline{p}(t)$ is obtained given $\underline{p}^*(0)$ the initial conditions, and so $\underline{p}(f)$ the state probabilities at the end of a system run are obtained. During a simulation of operation involving many system runs the state probabilities at the end of each run are obtained from the probabilities at the start of that run. This process can be carried out whatever the duration of each system run and prevailing conditions within it.

The initial conditions at the start of each run except the first are obtained from the final conditions at the end of the previous run. This relationship is determined by the repair (if any) which is carried out between each run. The discrete probability transitions that arise can also be expressed in matrix-form by a linear transformation though in practice this is unnecessary waste of storage space. Each different level of repair will be represented by its own repair matrix Ri, i = 0 to k. Ro represents the end-of-run repair and in many systems the intermediate states are left alone and only the total system failed state is repaired, in which case Ro is diagonal except for one off-diagonal entry. In many systems the top level of repair ensures that all deficient states are repaired at some time Tk. In this case Rk will contain a top row of 1's and 0's elsewhere. The capability to reflect probabilistic repair is inherent in these matrices. In a time simulation these repair transitions are carried out at various times.

Steady application of all the above relationships enables the state probabilities $\underline{p}(t)$ to be built up indefinitely once $\underline{p}^*(0)$ is defined. It is not usual for the exact pattern of future operational use of a system to be known in advance. Certain guidelines are often known and these usually formulate into set repeatable patterns of system run duration and repair times. Thus in theory it is possible to build up regular relationships between the state probabilities immediately before or after a particular level of repair is applied. These relationships can often simplify particularly if Rk guarantees complete repair in which case $\underline{p}^*(m.Tk)$ is fixed such that state 1 is set to 1 and the rest to 0.

As noted already the end product that is required from a system reliability analysis is not a state probability vector but a rate per hour, or similar units, of total system failure for a given system design and a given operating policy which includes run times and repair policies and intervals. If the n states have been ordered so that the "total system failure" state is state n (and they can always be so ordered) then it is the information within the time sequence pn(t) which is used to actually calculate this figure. The potential for analytic simplification is strongest in the equations governing the pn(t) sequence which usually has certain specific properties.

We may reasonably suppose that the initial value pn*(m.f) at the start of any system run is 0 which conveys the meaning that the repair policy ensures that any system run will not commence if the system is already totally failed. We may also reasonably suppose (by definition of the term "run") that a total system failure may occur within a run but that it will occur no more than once in any one run. We may further assume that if system failure occurs within a run that it stays in this state until the end of the run. These assumptions yield certain usable information. It means that state n is a "trapping" state during a run so that the probability of the system being in this state will never decrease during the run. It further means that at the end of the run the quantity pn(m.f) represents the probability of having had a system failure during that run. Hence the system reliability calculation comes down simply to taking the probabilities of system reliability calculation comes down simply to taking the probabilities of system failure in each run and obtaining a mean value over however many runs seems appropriate.

If the assumptions are relaxed so that it is possible to start a run with the system already in its failed state then we would not be able to assume pn*(m.f) = 0. But we have to ensure that in this case the system reliability calculation does not multiple account. That is to say, a system failure that may have occurred in a previous run must uot be counted again later. The equation for system failure rate is amended to take the initial probabilities in each run off the final probabilities. It is often the case that terms cancel within the summation leaving a summation based on probabilities at repair intervals rather than system runs. If state n does not "trap" then the calculation does become more complex and the system reliability calculation is not so simple. But even in this case it is to be hoped that the information in the Q and Ri, i = 0 to k, matrices will point the way towards the correct calculation.

The only remaining item to be cleared up is the choice of the simulation run-time T (or number of systems runs M) over which to perform the probability and reliability calculations. The hint as to what to do lies in the repeatability (if any) of the system run-time patterns and the repair intervals and in the nature of each repair level. If at some point in time complete repair is guaranteed and some statement on the frequency of such an event can be made then it can be said that the probabilistic determination of the system over one time period Tk is repeated for all such time intervals thereafter. The choice of T is obvious in this case. In an experimental situation where the effect of repair intervals on the total system failure rate is being investigated then whilst in general the simulation must be re-executed for each combination of repair intervals it is worth noting that some saving of effort is possible.

Returning to the main point the outstanding question in determining the choice of T is the possibility of faults in the system that are either dormant or are not repaired at any repair level except after a further transition to another state. If there is never a time at which full system repair is guaranteed (and with many complex systems this is often the case) then the choice of T becomes a lot less clear. Clearly putting $T = \infty$ is no more productive for a Markov time simulation than it is within a fault tree analysis. However it is almost certainly true that such faults will be repaired at some point in time, and those that are not might as well be treated as failed from the outset, so the equivalent of the fault-tree exposure time is not infinite nor is it necessarily related to system life.

The question is simply one of consideration of what happens to a system with dormant faults as time increases indefinitely. It would be natural to expect the system failure probability per run (after adjustment for the known repair interval considerations) to increase as time proceeds, but how far? Eventually a steady-state solution must be reached, at which the sequence $\underline{p}$* (m.f.) will settle out and become a fixed cyclic pattern based around the known repair intervals. If this cyclic pattern were to be used to set the initial conditions then the steady-state solution would be immediately obtainable. The steady-state solution is clearly the one which will yield the system reliability information, just as where there were no dormant faults the steady-state solution is obtainable from the first interval within which f ll repair is assured.

The steady state solution can be obtained if the system run-time and repair interval patterns are regular and if the matrices lend themselves to analytic solution. The sequence $\underline{p}$*(m.T_k) is then founded upon a constant linear transformational relationship and the steady state solution $\underline{p}$' is the eigenvector corresponding to the eigenvalue of 1. The nature of the component matrices that make up this matrix are such that the eigenvector-eigenvalue problem falls into a special case that avoids most of the awkward and complicated mathematics. The biggest problem is in obtaining the transition matrix for each system run and combining it with repair matrices in several matrix multiplications. Also because the matrices in question are sparse matrices it is not in general computationally efficient to store and calculate in this form.

Consider as a theoretical example to illustrate the process and pitfalls a very simple system which contains two boxes with failure rates r_1 and r_2 respectively and operation is possible if either box is operating. The four possible states are

1 Everything working.

2 Box 1 failed.

3 Box 2 failed.

4 Both boxes failed (total system failure).

The application of the mathematics to this system is added to Appendix A and serves to illustrate that the Markov method arrives at an exact theoretical result which is not obtainable by fault tree analysis.

3.3 MARKOV PROBABILITY ANALYSIS PACKAGE

A package has been developed at Rolls-Royce which handles the Markov Probability Analysis (MPA) of any given system in the way described above. the calculations are done numerically at this point in time, no advantage being yet taken of any theoretical results that could simplify the calculations. The information on which Q, R_i, i = 0 to k, are based and the quantities f, T_i, i = 1 to k, are entered at the keyboard or from a data file containing the Markov model. The matrix information can also be read from a transition table data file as generated by the RMG package and default values are then initially assigned to the calculation and output control parameters. All variables are interactively changeable and various forms of output obtainable. After execution the model and calculated data can be stored in data files and/or printed out.

The solution is obtained during each system run by an appropriate integration method applied to the basic Markov differential equation. This method could be a linear multi-step method, or a Runge-Kutta method, but the method so far implemented is the standard Euler method (see for example reference 7 for all these definitions). The rather marginal stability properties of this rule are nullified by the absolute convergence properties inherent in the matrix Q. The information on repair is stored in vector form rather than the matrices R_i, i = 0 to k, but the effect is still the same. The simulation package also requires the initial conditions $\underline{p}*(0)$ to be set up. The simulation is then executed for a specific length of time and output of numerical and/or graphical data is produced.

The example from reference 2 is used to demonstrate the execution of the MPA package. The system is as shown on figure 1 and contains 6 sensors reading 3 different inputs and supplying the inputs to 3 identical computers in the manner shown with inter-computer communication. Each computer performs a calculation based on whatever inputs it has available either directly or via other computers. Each computer requires values from each of the 3 inputs to be validated by cross-check between the 2 readings of that input or by model-check from the other inputs, otherwise the computer will give a null output to the actuator. The actuator averages the non-null outputs from the computers. There are 2 types of failure these being the sensor failures each with a failure rate r1 = $0.2 * 10^{-4}$/hour, and the computer failures each with a failure rate r2 = $1.0 * 10^{-4}$/hour. For a fuller description of this system see reference 2.

The Markov model was read into memory from a transition table data file previously generated and created by the RMG package. The Markov states and transitions are shown graphically on figure 2 with the states and the differential equation listed on table 1. The error in reference 2 was that states 4 and 5 were considered to be the same. Appendix B shows a typical print-out obtained from executing this model under the auspices of MPA. The displayed data is split into 5 sections each of which is optional and these are

1) Model information relating to state transitions and rates.

2) Operating conditions information relating to initial probabilities, repair policies and intervals, and time base variables.

3) Output specification information relating to the combination of states into outputs and the printing/numerical/graphical output control variables and scales.

4) Tabulated data of state probabilities.

5) System reliability data.

The output was so arranged to display the probabilities of the mutually exclusive and complementary variables

a) Everything working.

b) All intermediate states combined.

c) Total system failure.

at the end of each system run of duration 12 hours. The system reliability information yields various information among which is the rate for total system failure over the specified simulation execution period which works out to be $3.955 * 10^{-6}$/hour. Figure 3 shows the corresponding plot to the tabulated data of Appendix B with just outputs b and c displayed each on its own axis and with its own line identification. Whilst figure 3 links up the end-of-run probabilities, figure 4 plots out the data for the same example at each calculation point (0.1 hours in this case). The start-of-run probability of total system failure is always 0 and the end-of-run probability increases steadily. The end-of-run system failure rate eventually reaches steady-state. Figure 5 shows the same plot as figure 3 on a 100 times greater time base and 20 times greater probability axes. When steady state is reached it can be seen that the average system failure rate is 0.017 failures per hour with 65% of all system runs commencing with at least one fault already in the system.

4 RELIABILITY MODEL GENERATION (RMG)

Whilst the Markov Probability Analysis method has been seen to eliminate many of the errors and omissions inherent in traditional reliability methods and all of the arithmetic approximations, it is still true that errors and omissions can occur. The trade-off of visibility and complexity versus assumptions and approximations can lead to errors which are not so likely with the Markov method but still possible.

The fundamental reason for such errors and omissions lies in the fact that a reliability model needs to be available in some form before it can be transferred into the Markov form or into a traditional form. This means to say that the reliability engineer needs to determine the system failure causes and other associated information before drawing a fault tree or writing down Markov states. It has been shown already that the Markov method reduces the problem to consideration of single faults applied to individual states rather than an ad-hoc formulation of system cut-sets, the former process being less prone to errors. However, it has also been shown how it is possible to make assumptions that are not true in setting up the Markov states even for not-so-complex systems. The example of reference 2 contains 6 hardware parts which are identical with each other, 3 other identical parts, and an immense amount of system symetry. This positively invites the temptation to set up a simplified set of system states. The only way to truly ensure that the set of states is correct is by the painstaking approach of generating all the possible configurations in which the system may operate and then to reduce the number of configurations systematically until a residue of states is found. This sounds worse than it is as numbers of configurations is theoretically exponentially related to the number of distinct parts considered but by taking system information into account yielded for the reference 2 example only 39 working configurations and these then reduced to the residue of the 9 states of table 1. The processes described here could be carried out manually but any significant system complexity would soon render the task very laborious and liable to error simply out of sheer monotony.

The automation of the painstaking method of generating states and transitions is what the Reliability Model Generation (RMG) method is all about. It is about the transfer of fundamental system information, carried either in the engineers mind or on sheets of paper, into a reliability model using the rules that are already inherent if the process is done manually. At first sight it seems mind-boggling to get a computer to interpret system design but it will be shown that such a formulation can be achieved and that the system design information does permit breakdown into certain well-defined categories.

With the Markov approach to reliability analysis the problem of advance definition of the failure cut-sets is already reduced to that of definition of system states. With reliability model generation the problem of advance definition of system states is further reduced to that of definition only of the set of single faults and repair transitions that can affect the system.

4.1 MODEL GENERATION CONCEPTS

There are 2 stages to the reliability model generation mechanism these being

a) the generation of all the possible operating configurations

b) combination of configurations by equivalence into a residue of states

The first process involves keeping an index of configurations as they are discovered, a record of the sets of faults within each configuration, and a transition table each line of which lists the outgoing transitions from each configuration. In the reference 2 example there are 9 distinct hardware parts and model generation is based around 9 fault events. Table 2 shows the manual application of the model generation concept to this example and the resulting transition table. It can be seen that the process is built up from configuration 1 (the "everything working" configuration) by systematic application of each fault event to each configuration until all configurations have been analysed in this way. The system information is used to determine whether system operation is still possible there being no need to consider configurations in which total system failure has occurred. Repair transitions may or may not be analysed in parallel with the fault transitions.

If there are v fault events that can occur within the system then these events will form the bulk of the set of events to be applied to each configuration. If it desired to analyse repair transitions also then assuming end-of-run repair is definable and also k further repair levels are assumed then k+1 repair transitions are also applied to each system configuration. So the length of each line of the transition table is either v or v+k+1 depending on whether repair is being incorporated or not.

The transitions caused by the v fault events correspond to the entries in the Q matrix. The transitions caused by the k+1 repair events correspond respectively to the entries in each of the repair matrices Ri, i = 0 to k or to any representation in vector form. The application of each fault to each configuration is the process which a reliability engineer could do manually using knowledge of the system. The application of the repair policies is simply that of interpretation of each repair level in turn to the set of faults within the configuration under consideration to determine if that configuration is repairable at that repair level. To perform this process automatically requires certain information to be entered into the RMG package in advance of model generation which will then be processed in a way which will achieve the correct transitions. The information required is described in section 4.2. The primary task remaining is to define the set of single fault events that can occur.

It is possible for a complex system to have within it a multitude of components. However, it is also likely that many of these components will have exactly the same effect on system operation. It is therefore entirely reasonable to treat such components within a reliability analysis as if there were instead one component with a failure rate made up of the total of the rates of failure of the individual components. This collection of components classified by system effect is known as a "block" within the reliability model.

Very often a system is designed on a modular basis and this translates, in reliability terms, into the fact that all the components in one module could have the same failure effect on the system. Thus a hardware module and a reliability model block could represent one and the same thing. Even where a hardware module contains components with different system effects, it is often possible to consider a "worst-case" scenario in order to combine the components into a single reliability model block. This would have the effect of introducing a conservative assumption and any model generated with such an assumption would give a higher figure for total system failure than the true answer. These assumptions are best avoided if possible. To represent reality it is necessary to separate components strictly by system effect. But a complex system many necessitate some combinations as above. It is important to stress here that these assumptions are conservative ones, and as such lead to an upper bound for the total system failure rate, and that this is the correct way round, rather than to consider system failure sets and risk omissions.

The entire system is classified into these various "blocks" according to system effect. So what is meant by system effect in this case? It is concerned with such things as fault-propogation, operational logic, repair logic, and fault types, and various other concepts which become clearer when the specific information in considered. Fault-propogation basically is concerned with the effect of faults in one block upon other blocks. Operational logic is concerned with what hardware needs to be operating in order to maintain system operation. It must be stressed again that system design is usually related to this requirement rather than to the failure cases, and in any case omissions from the operational logic are conservative. Repair logic states the repair policy in terms of the model blocks and is therefore classified as a system effect. Fault types considerations is a recognition that within a block, within a component even, failures can exist which exhibit different failure modes and such failure modes are therefore, if not already separated by system effects, classified by detection methods. The total of all the fault types within all blocks forms the set of v fault events required for the model generation process.

So given these concepts which should theoretically define the reliability model blocks, the real question is how the blocks are defined in practice. Inevitably attempts will be made to tie the blocks in with the hardware modules, and it is already recognised that this can and should be done on a conservative basis, so that any fault within a block will cause the entire block to be deemed as having failed. The system design assumptions are the starting point for block definition, particularly if this is modular. The system design will provide information in the required categories. After this the Failure Modes and Effects Analysis (FMEA) for the system or for the modules of the system is brought into play. Within the FMEA every possible single fault is (or should be) listed there, and certain information is provided that is used to classify the faults within the reliability model. The FMEA(s) will usually have the effect of taking the block definition away from its modular starting point.

The important thing to note here is that the combination of system design assumptions and the system FMEA(s) are used to make the reliability model steadily less conservative and more real. The ultimate starting point is to assume that the entire system can be classified into just one block. This represents a return to the non-redundant systems mentioned right back in section 1. Use of the system design assumptions and FMEA(s) then help to break the system down into more and different blocks to get as near to reality as desired. Providing that each breakdown is not contradictory to the system design assumptions or FMEA(s) then the resulting reliability model is guaranteed to provide an upper bound for the system failure rate.

Once the blocks are fully defined, and the fault types within each block are fully defined, then the total number of fault types summed over all the blocks represents the number of different, mutually exclusive and mutually complimentary, single faults that can occur. These are the fault events that are applied to each generated configuration. In the reference 2 example there were considered to be 9 blocks exactly corresponding to the 9 hardware parts as all faults in any one part yielded one specific system effect. Similarly all 9 blocks had just 1 fault type each as there is no distinction within any hardware part in respect to methods of detection of the faults within that part.

Having defined the blocks and fault types and set up the system information based around these blocks then configuration generation takes place either manually or automatically using the RMG package. The second part of the process involves reducing the configurations into a residue of states. This is done by comparing lines on the transition table and determining if any 2 lines are equivalent to each other. It must be stressed that even where 2 configurations are similar in terms of containing equivalent hardware that those configurations are not necessarily equivalent in terms of outgoing transitions. The hardware equivalence within the system is used to help determine transition table equivalence but is not to be confused with it. Table 3 shows the final transition table for the residue of states after equivalencing table 2. Note that states 4, 5, 6 all contain equivalent hardware but their transition table lines are not equivalent. These states correspond to the definitions on table 1 and every one of the 39 configurations on table 2 fits into one of the states.

4.2 DESIGN INFORMATION DATA

The design information is that which is required in advance of carrying out the model generation process. This information is that which a reliability engineer uses on an implicit basis to effect either failure cut-sets or Markov model states or, in this new formulation, the generation of configurations from a given block structure. In order to automate the process each piece of design information needs to be formulated and will be illustrated using the same example so far considered.

So far there are 9 distinct types of design information that have been automated although it is not claimed that this formulation is unique or necessarily complete. The 9 types are

1) Block Structure.

2) Fault Propogation.

3) Operational Logic.

4) Repair Logic.

5) Fault Types.

6) Detection Methods.

7) Equivalence of Hardware.

8) Dormant Logic.

9) Fault Count.

The first information that is required by the RMG package is the block structure. This can be done quite flexibly, so as to take into account the possibility of a modular division among the parts of the system. For the given example a block structure of 9 top-level blocks is the obvious starting point although in practice such a system may yield further breakdown of the computers or include faults in the highways or whatever. If it was considered that the 9 blocks represent a conservative situation and the computers, for example, are found to have a non-simple reliability analysis of their own then this can be structured by retaining the 9 top-level blocks and creating sub-levels to handle any further sub-division of faults. These sub-levels constitute reliability sub-models the top event of which feeds into the next higher sub-model. The real benefit of multi-level block structures comes out in cases where the reliability model for the whole system is built up from the reliability model of its component modules. There is much flexibility to the method and for now it will be assumed within the example system that just one level of model exists with 9 blocks. The blocks are named on table 4.

The second information that is required by the RMG package is the fault propogation information. This information is specified separately for each and every sub-model within the block structure. It consists quite simply of a list of propogation effects which are put down until all propogation effects that can happen within the sub-model are completely specified. Each specified propogation effect takes the form of

a) A logic string specifying a fault combination which causes the propogation effect.

b) *Indication of* whether fault is real or imaginary.

c) A list of blocks which are affected by the propogation.

The logic string can be made of 'AND' and 'OR' gates between indices of blocks within the sub-model. The indication of whether fault is real or imaginary is intended to handle 2 different types of fault propogation. Sometimes, e.g. electrical power faults, the failure of other hardware parts is real. In other cases, e.g. where an input sensor reading is fed to 2 different places, the fault propogation effect reflects an equivalent system effect. In these cases the existence of the propogation effect enables the operational logic to be simplified and also reduces the number of working configurations generated. The lists of blocks affected speaks for itself. The reference 2 example has 3 propogation effects within its one sub-model. These effects specify that if one of the 3 computers fails then the 2 sensors which are fed into the system through that computer (figure 1) are no longer accessible to the system. Table 5 shows the full list.

The third information that is required by the RMG package is the operational logic. This is also entered separately for each sub-model and specifies for that sub-model the combinations of blocks which need to be operating in order for the sub-system described within the sub-model to continue to operate. This is the right approach as the more combinations that are added to the operational logic the nearer the model will get to representing reality but at the same time ensuring that reality is approached from the conservative side by only adding any particular combination to the operational logic string when it is established that operation is possible with this combination. The operational logic that may be entered can handle 'OR' and 'AND' logic. It can further cover a type of 'NOT' logic that allows different operational logic combinations to be considered in the event of specified faults having already occurred. This is particularly relevant to software-based systems which in the event of faults within the incoming data can quickly reconfigure to do entirely different calculations using an entirely different set of incoming data. It is also applicable to alternative power supply type situations where electrical hardware parts may be depowered and then powered-up again. In the current example it will be assumed that no such alternative control modes are available. The operational logic for this system is based around the requirement to have certain sensor readings available to the system. It becomes apparent when studying the design assumptions that the system will operate if there is at least one sensor of each input type A, B, C or if there are 2 pairs of similar sensors available. It is apparent that if enough sensors are available to the computers then it is certain that there are enough computers to process them and the fault propogation logic has ensured that the system reflects *this fact*. The logic string for which it is known that system operation can take place is given on table 5.

The fourth information that is required by the RMG package is the repair logic. This information is concerned with stating the assumed operational repair policies in terms of the defined block structure. This is not done on a sub-model basis as repair policies are in general formulated, or merely just happen, on a system-wide basis using the low-level blocks. The low-level blocks are those for which no further sub-model exists so that in the example system there are 9 low-level blocks. This logic is, in essence, of contrary nature to the remainder of the design information as it alone is concerned not with what can go wrong within the system during a system run but with what can be put right between each run. It is often desirable to compare several repair policies in order to determine effect on the system reliability figures. There are 2 ways of making such a comparison

a) Generate the model without reference to repair transitions and then determine manually the application of each repair policy to the generated model states and enter into the MPA package.

b) Generate separate reliability models for each different repair policy thus making each model a unique combination of system design and associated repair policy.

The number, k, of scheduled repair levels is required followed by a logic string for each of the k+1 potential repair transitions corresponding to the information within the matrices Ri, i = 0 to k. Again 'AND' and 'OR' logic is used for the logic at each repair level and again approach from the conservative side is assured as each logic string is built up. For the current example no repair assumptions are made except that of specifying that total system failure is repaired at the end of the run in which it occurs which is already inherently assumed in the RMG package. Thus for this model repair considerations can be omitted altogether as on table 2. If the repair assumptions were specified it would be done by putting k = 0 and specifying the logic string for repair level 0 as blank. This would simply add one column to table 2 and every entry would be a '.' and so the equivalence of lines on the table would be unaffected.

The fifth information that is required by the RMG package is that of fault types within each block. This information specifies for each low-level block within the system what further sub-divisions exist due to existence of different methods of detection. Any fault within a block will cause that block to be deemed as failed but whether the failure is known or unknown, and hence accommodated according to the operational logic alternatives, depends on the method of detection and whether that detection is working successfully. Thus each fault type within each block is a separate fault event as the system effect is potentially different in each case. The information entry that is required is simply to state the number of fault types in each low-level block in turn. The total failure rate for each block is divided among its fault types and the contributing component faults are classified in these fault types. The total number of fault types over all the low-level blocks within the system defines the total number v of fault events to be applied as potential transition events from any one state into other states. In the current example with just one sub-model and 9 low-level blocks it is declared that each block has just one fault type giving a total of 9 fault events as shown on table 2.

The sixth information required by the RMG package is the detection methods for each fault type within each block. Each detection method is expressed as a logic string which states which block or combination of blocks needs to be operating at the time the fault is applied to each configuration in order for the fault to be successfully detected. If the required block combination is not available at the time when it is needed then the fault will be registered as undetected. Again 'AND' and 'OR' logic to set up the detection method combinations for each fault event. The current example has 9 fault events. The detection method philosophy states that each sensor failure can be detected by its own computer and each computer failure is detected by itself. The sensor failure detection is thus always certain as the fault propogation logic insists that if a computer is failed then so are both its sensors and the issue of future sensor failure would never arise. The computer detection method is another way of saying that the system will always detect computer faults, due in this case to the actuator ignoring any null output. The logic strings for the detection methods are as on table 4.

The seventh information required by the RMG package is the statements on equivalence of hardware. This is entered for every fault event within the system and specifies that each fault type may contain identical hardware parts to another fault type elsewhere in the system and thus have the same failure rate. It has been noted already that this type of information about the symmetry of a system in hardware terms is often mistakenly used to assume that 2 or more system configurations with simi`ɔr hardware faults are equivalent configurations and can be combined into the same state of the reliability model. The true definition of equivalence of configurations lies in the various transitions from the configurations. It is in determining equivalence of lines of the transition table that the hardware equivalence is taken into account but not in the generation of configurations. The hardware equivalence in the current example concerns the fact that each sensor is hardware equivalent to each other sensor and similarly for the computers. The logic entry is that each sensor is made equivalent to sensor XA and each computer made equivalent to computer X as shown on table 4.

The eighth information required by the RMG package is the dormant logic which is concerned with whether faults which are undetected during a system run may beco e detected between system runs. If some unknown faults do become detected in this way then they can be considered upon application of the repair logic. If repair is not applied then the next system run starts off in a different configuration which may or may not be in same state. Thus such considerations lead to potential discrete state transitions which would also be reflected in the matrices Ri, i = 0 to k, along with the repair logic. The information is done on a low-level block basis and is entered simply as a 'Yes' or 'No' to the question of whether the failure of each block in turn could be so detected. In the current example there is no possibility of undetected faults in any block and the answers to the questions are irrelevant and table 4 shows that 'No' is answered for each block.

The ninth information required by the RMG package is the fault count logic. This concept is not specifically related to system design assumptions and is optional. It is intended to assist computation by trying to find the optimum balance between numbers of configurations generated and the amount of conservatism built in by reducing such numbers of configurations. The concept arises out of a possibility that the RMG package could generate a great multitude of configurations in which the system could continue to operate, many of which would correspond to multiple-fault cases and be very much less likely to occur than configurations with fewer faults. It is often justifiable to conservatively set a cut-off point by collecting the high fault-number configurations together within the state corresponding to total system failure. So during the model generation process a fault count limit may be set at which system failure is assumed. The information that is required during the design information data entry is to set a count to each low-level block within the system. If this block then fails during a system run the appropriate fault count is added to the 'score' until the specified limit is reached. This count can be different for different blocks and can be 0 as well as any positive number. In the current example it is proposed not to consider limiting faults in this way and so the count for each block is set to 0 as shown on table 4.

4.3 RELIABILITY MODEL GENERATION PACKAGE

A package has been developed at Rolls-Royce plc which handles the Reliability Model Generation (RMG) of any given system starting from the entry of the design information as described above. The information is entered at the keyboard or from a design information data file. This information is then coded up into a form which will enable rapid computation during the model generation process. When the design information is fully entered then it may be interactively altered and written, if appropriate, to a design information data file. When the information is deemed correct then the model generation can take place. This is done firstly by generating the various system configurations that still allow system operation.

The index of operating configurations is initialised with configuration 1 which is the fully-operative no-fault configuration. This configuration is then analysed by application to it of each of the v fault events separately (and each of the k+1 repair events if selected) and several more operating configurations are discovered and added to the index of such configurations. The first line of the transition table is made up of the indices of the configurations resulting from transitions that take place on occurrence of the various fault and repair events. From then on each configuration in the index list is analysed for outgoing transitions in exactly the same way, the lines of the transition table are built up, further configurations added to the list when discovered, until all configurations are analysed in this way.

At the application of each fault event to each configuration certain operations are carried out. Firstly if the block corresponding to the fault event is already in a failed condition within the configuration then this event cannot be applied again and the transition table entry is blank to indicate no change in the configuration. Assuming the fault event is a new one to the configuration under consideration then the detectability of the fault is evaluated from the detection method and the existing faults within the configuration so that the block is flagged up as failed either undetected or detected. The fault-propogation effects are then employed to pass the fault onto other blocks within the sub-model if appropriate and following this the operational logic is evaluated to determine if the sub-system defined by the sub-model is still operating or has failed. If it has failed then the fault is passed onto to the next higher level sub-model and the process repeated until total system operability is determined. If the system has failed so that operation is not possible then the transition table entry is set up to flag this fact. If the system is still operating then the new configuration will need to be added to the index of operating configurations unless the configuration is already in this list and the list is checked to see if this is so. Either way the discovered configuration will have or be given a place in the list and the transition table entry references it.

At the application of each repair event certain operations are carried out. Firstly the dormant policy is applied to the undetected faults to see if they are to be transferred to the status of detected faults. Following this the repair policy corresponding to the repair level under consideration is applied to the detected faults to determine if repair takes place at this level. If repair is to be carried out on this configuration at this level then the transition table entry shows a transition to the "everything working" configuration. If repair is not to be carried out and no undetected faults have become detected faults then the transition table entry is blank to show no change in configuration. If there are some undetected faults becoming detected then the configuration list is checked exactly as for the fault events.

When the configurations list and transition table is complete then the process of identifying equivalence starts, but this process can also take place at intermediate times if it is desirable to cut down on memory storage and list checking times during the configuration generation process. The process of equivalence involves comparing lines on the transition table taking hardware equivalence into account. If 2 lines are identical then the configurations are deemed equivalent and the configurations list is reduced by one for each such equivalence obtained. This process is iterative as the equivalence of some configurations will usually lead on to the equivalence of others. When all the equivalence relationships are carried out then the final residue of configurations and the corresponding transition table represent the minimum number of states of the reliability model. These will be the states that are used in the Markov probability analysis. Thus in the final form of the reference 2 example the transition table is 9 by 8 for the 9 fault events (there are no repair considerations) and the 8 operating states.

The final residue of states and transitions in this form can then be put onto a transition table data file ready for use by the MPA package.

To further illustrate the process let us amend the reference 2 example model slightly in 2 ways. Firstly let us apply a repair policy that specifies 2 levels of scheduled repair. At level 1 repair takes place if any computer is failed, and at level 2 repair takes place if any sensor has failed. The design information data entry is altered only at the repair logic stage. Here k = 2 is stated and 3 logic strings are entered. The first string remains blank, the second is an 'OR' combination of the 3 computers, the third string is an 'OR' combination of the 6 sensors. Table 6 shows the generated transition table and also the residue of states. The 39 configurations are unchanged from table 2, the transition table entries are augmented by the 3 repair levels, and when equivalencing is done it is found that there are now 2 extra states. This model can be executed using the MPA package and the reliability is improved by the introduction of a repair policy.

For a second alteration let us return to the standard model and forget repair considerations. Instead we shall suppose that each computer is dual-redundant and is thus made up of 2 sub-computers only one of which needs to operate to allow an output from the computer to go to the actuator. The design information is altered by the addition of 3 sub-models corresponding to the sub-division of each computer. Within the new sub-models the operational logic is a simple 'OR' statement between the 2 sub-computers and there are no fault propogation effects within these sub-models. The top level sub-model remains unchanged. The fault types and detection methods are extended to the new set of 12 low-level blocks in the appropriate manner, and the sub-computers are all deemed to be hardware equivalent. Table 7 shows the first 8 lines of a transition table which extends to 891 configurations, and also the final transition table based on 38 working states. Just a slight model complexity has produced a significantly increased model but presents no difficulties for the RMG package.

5 TOTAL ANALYSIS SYSTEM AND EXAMPLE OF APPLICATION TO A CIVIL TURBOFAN ENGINE CONTROL SYSTEM

The operation of the total package contains various data files and keyboard, screen, and printer possibilities as previously described. A typical 'minimum' order of operation of user selected activities is as follows

1) Start executing RMG.
2) Keyboard entry of design information data set.
3) Model generation.
4) Write out to a transition table data file.
5) Start executing MPA.
6) Read in transition table data file.
7) Keyboard entry of fault event failure rates.
8) Time simulation.
9) Display and/or print and/or plot probability and reliability data.

There are 4 different data file types currently available within the package

a) Design information data files.
b) Transition table data files.
c) Markov reliability model data files.
d) State probabilities data files.

These files reflect the common desire to break off and resume computation or to alter the computation from a particular point without in either case having to start from the beginning. For instance, it may be desirable to compare different policies with the same design assumptions, or it may be desirable to run a complete model with different repair times or with different failure rates or with different variables tabulated or plotted, or it may be desirable to store state probabilities for computing to a longer execution time without the need for repetition. Also at various points the data is inspectable and allows alterations to be made until the inspected data is what might be expected. This package is now used to determine the reliability of a dual-redundant electronic control system of a civil turbofan engine.

A typical civil turbofan engine is made up of a number of rotating shafts, each with a compressor and turbine, a combustion chamber for fuel entry and burn, and a by-pass system. The control system of such an engine has to ensure that the supply of fuel to the combustion chamber is correct taking into account the various measurements that can be taken from the engine, and it has also to ensure that the variable geometry of the engine is also appropriately controlled. Thus such a system will read measurements from the engine via transducers, translate these measurements into requirements for the controller functions, and then operate actuators to translate the requirements into physical reality. Such measurements as can be made typically include the rotational shaft speeds, the pressure and temperature of the air flow at various parts of the engine, feedback measurements indicating the results of actuator movement such as fuel flow rate, various on/off signals, and a measurement that conveys the pilots wishes for the power to be delivered by the engine.

Typically, nowadays, the control calculations are done within an on-board computer. Also not unusual nowadays is to build redundancy into the control system by having, for example, 2 identical lanes of electronic hardware, each of which is separately capable of engine control, and with an inter-lane highway to allow cross-channel communication so that it is not necessary to change an entire lane for every fault. These 2 identical lanes contain circuits associated with a computer, inputs, outputs, and power supply. The control system is known to be capable of containing to operate if it has at least one of each function within the dual-redundant parts still operating, and if the inter-lane highway is operating whenever it is needed for cross-lane communication, and if the non-redundant parts associated with actuators are operating. The power supply in each lane is itself dual-redundant with a primary and secondary source either of which can operate the electrical functions of one lane.

Electrical faults have much potential for fault propogation. Within the analysed system it is known that total power to one lane will fail all the inputs, outputs, and computer of that lane, also it is known that the computer provides a clock signal to certain inputs, also it is known that certain conditioning circuits cause loss of more of than one input or output. Outside the dual-redundant parts it is known that certain input functions are affected by common-mode hardware which can cause loss of input to each electronic lane, also it is known that some faults exist which will cause total system failure either directly or through fault propogation effects.

The operational logic is based around the information above concerning dual-redundant hardware and the inter-lane highway and taking the fault propogations into account. Further design information is known about the software calculations done within the computer. These calculations implicitly divide the inputs into 3 types

a) "Don"t care" inputs which are not used in the control calculations.

b) "Accommodated" inputs which are used for primary control calculations but for which a set of back-up reversionary calculations exist if these inputs are unavailable.

c) "Vital" inputs which are used for primary control calculations and which cannot be accommodated.

The system will operate a "preferred" order of active control hardware. Lane A hardware is preferred to lane B's wherever it is available and complete lane change only occurs if power to lane A is lost or if lane A and the inter-lane highway are both faulty. The system will also "prefer" primary control mode to reversionary mode.

The repair philosophy is reasonably sophisticated. It is based on 2 scheduled repair levels plus end-of-run repair. The end-of-run repair criterion is that if power is completely lost to one lane or if there are faults in any input or output and inter-lane highway together then repair must be made before next run commences. The level 1 repair covers loss of either power supply in either lane and also the total loss of any input or output function. The level 2 repair covers loss of any input or output or the inter-lane highway, that is to say every known fault is repaired at level 2.

Many faults are detected within the software of either computer with the continuing proviso of inter-lane highway operating if one computer is to detect input or output failures in the other lane. Some faults are detected within the software but with the further requirement that a cross-check is made with the same input or output in the other lane which then also needs to be operating. Computer and power supply faults are all reckoned to be automatically detected within the system hardware. Some input faults are actually believed to be undetectable of which such faults in "don't care" inputs do not matter but others would. The inter-lane highway failure is detected by either computer being able to make comparisons between the inputs and outputs of each lane.

The equivalence of hardware is described simply in that the dual-redundant hardware of lane B is reckoned to be exactly identical to that of lane A. The dormant logic will assume that no detection capability exists between system runs.

All of the above design information guides the choice of block structure and then sets the various logic information once the block structure is settled. The natural block structure to adopt would be to have individual blocks for each input and output and computer and power source within each lane and for each non-redundant function also. For demonstration purposes this creates too many blocks to be readily digested. So the various functions need to be combined into a smaller number of blocks, which builds in some conservatism, there being no need to make the analysis any more complex if this is achieved.

The chosen block structure has been to create a top level of 16 blocks of which there are 6 in each lane covering the "don't care" functions, the "accommodated" functions, the "vital" functions, the "multiple" functions (those affecting multiple inputs and outputs), computer, and power, and 4 others covering common-mode "don't care" functions, common-mode "accommodated" functions, single faults causing system failure, and the inter-lane highway. The power supplies are then split into 2 sub-models each with 2 blocks which cover the alternative power sources. This gives a total of 18 low-level blocks and the various information relating to each of these low-level blocks is shown on table 8. There are 25 fault events and 3 repair events to be analysed in the generation process. There are just 15 failure rates needed and these are also given on table 8. The fault propogation and operational logic for each sub-model and the system-wide repair policy is given on table 9.

An analysis is carried out which shows the effect of variation of the conservative assumption built into the fault count limit. Table 10 gives the results for the numbers of configurations, numbers of states, and total system failure rates against the fault count limit. The table shows that when the fault count limit is set at 1 then every defect in the system leads to total system failure, and that when the fault count limit is raised the total system failure rate drops and homes in onto the rate that is achievable if the fault count limit was not applied. This is seen to happen at an increasing cost in number of configurations, but the number of states increases firstly and then drops very rapidly. This latter effect is due to the fact that system equivalences which have been suppressed by an arbitrary fault count limit are now being seen. These results assume a system-run (flight) time of 10 hours, and repair times of 50 and 400 hours.

An analysis is also carried out to show the effect of varying the repair times. Table 11 gives the results of such variations using the fault count limit of 4. It can be seen that for this system and this assumed repair policy that the variation of the level 2 repair time is more significant than that of level 1. The results include the ultimate system failure rates as the repair times are stretched to infinity so that only the level 0 repair is still carried out.

If a target total system failure rate of 10 per million hours is required then clearly the repair level times can be picked to achieve this. If no satisfactory combination of repair times and system reliability is obtainable then it is necessary to either alter the repair policy itself, or if this is not satisfactory, to expand the design information data set to remove more conservativeness from it, or if this doesn't work then the design cannot meet the requirements anyway!

6 CONCLUSION

A general philosophy has been demonstrated which has as its aim that of eliminating errors, omissions, and assumptions that can occur within a reliability analysis of fault-tolerant systems. It has been seen that by consideration of the effects of single faults on system configurations rather than by considering fault sets or sequences that these problems are somewhat reduced and can be eliminated altogether. A concept has been devised which seeks to focus the reliability engineers mind in such a way that all faults within a system are at least considered, and in a way that can be closely linked to the system design assumptions and checked from the FMEA. It is also carried out in a way that ensures overestimation, rather than underestimation, of the total system failure rate while the design assumptions are being formulated. It has been shown that the process of reliability analysis can then be automated right through to the final results, and a package illustrated which is user-interactive for flexible manipulation of the analysis or the output from it. The process as demonstrated is very thorough in its treatment of the faults within the system. There is further potential for use or development of mathematical results in order to simplify or speed up the calculations.

7 REFERENCES

1) Reliability Engineering and Risk Assessment. Henley and Kumamoto. Prentice-Hall 1981.

2) Full Authority Fault Tolerant Electronic Engine Control System for variable cycle engines. AF-WAL-TR81-2121. W Davies (Pratt and Whitney) December 1981.

3) Reliability Analysis of a Dual-redundant Engine Controller. SAE Aerospace Congress and Exposition. Anaheim, California. October 5 to 8 1981.

4) The SURE Reliability Analysis Program. NASA technical memorandum 87593. Ricky W Butler 1986.

5) Upper and lower bounds for semi-Markov Reliability Models of Reconfigurable Systems. Allan L White. NASA CR-172340 1984.

6) Reliability bounds for Fault Tolerant Systems with competing responses to component failures. Larry D Lee NASA TP-2409 1985.

7) Computational Methods in Ordinary Differential Equations. J S Lambert Wiley.

8) Control System Design. An introduction to state-space methods. Bernard Friedland. McGraw-Hill 1987.

APPENDIX A MARKOV MATHEMATICS

The basic state-flow Markov equation is given by

$$\dot{\underline{P}} = Q.\underline{P} \tag{1}$$

where $$\underline{P}\ (t) = (P1\ (t),\ P2\ (t),\ \ldots\ldots\ldots\ldots,\ Pn\ (t))^T \tag{2}$$

The solution of equation (1) is given by (reference 8)

$$\underline{P}\ (t) = e^{\int_o^t Q\ (t)\ dt} . \underline{P}^*\ (o) \tag{3}$$

where $\underline{P}^*$ (o) is the state probability vector at the start of the system run. We may define

$$W\ (o,\ t) = e^{\int_o^t Q\ (t)\ dt} \tag{4}$$

and substituting into (3) gives

$$\underline{P}\ (t) = W\ (o,\ t)\ .\ \underline{P}^*\ (o) \tag{5}$$

let us suppose, for conceptual ease only, that Q(t) is a constant matrix Q so that W (o, t) becomes W (t). Then suppose also, again for conceptual ease only, that each system run is of constant duration so that (5) may be generalised as

$$\underline{P}\ (m.f) = W\ (f)\ .\ \underline{P}^*\ ((m-1).f) \quad m \geqslant 1 \tag{6}$$

The relationship between $\underline{P}^*$ (m.f) (state probabilities at start of run m+1) to $\underline{P}$ (m.f) (state probabilities at end of run m) is determined by the repair (if any) which is carried out between the 2 runs and by any other between-runs effect, e.g. detection of unknown faults, that can occur between runs. In general there are some discrete transfers of probability at such times that can be expressed as a linear transformation given by

$$\underline{P}^*\ (m.f) = Ro.\ \underline{P}\ (m.f) \quad m \geqslant 1 \tag{7}$$

where Ro will usually contain 0's and 1's only, the exception being in systems where probabilistic events occur between system runs. If we now assume that in addition to the end-of-run repair transitions there are also k further repair levels that the system is operated to at various scheduled intervals Ti, i = 1 to k, then these are expressed as

$$\underline{P}^*\ (m.Ti) = Ri.\ \underline{P}\ (m.Ti) \quad i = 1 \text{ to } k,\ m \geqslant 1 \tag{8}$$

again assuming *constant* repair intervals. Now let us suppose, for conceptual ease only, that the repair level time interval T_1 is made of exactly M_1 runs of duration f. Then combining equations (6), (7), (8) appropriately gives

$$\underline{P}^*\ (m.T_1) = R_1.W(f).(Ro.W(f))^{M1-1}.\underline{P}^*\ ((m-1).T1) \quad m \geqslant 1 \tag{9}$$

Now if we define

$$\begin{aligned} W^{(0)}(f) &= W(f) \\ W^{(1)}(T1) &= R1.W(f).W^{(0)}(f)^{M1-1} \end{aligned} \tag{10}$$

then substituting (10) into (6), (7) and (9) gives

$$\left.\begin{aligned} \underline{P}^*\ (m.f) &= W^{(0)}(f).\ \underline{P}^*\ ((m-1).f) \\ \underline{P}^*\ (m.T1) &= W^{(1)}(T1).\ \underline{P}^*((m-1).T1) \end{aligned}\right\} m \geqslant 1 \tag{11}$$

Clearly (11) suggests a generalisation to the known k levels of repair and this is evaluated as

$$\left.\begin{aligned} W^{(i)}\ (f) &= Ri.\ W(f).\ W^{(i-1)}(f)^{Mi-1} \\ \underline{P}^*\ (m.Ti) &= W^{(i)}(Ti).\ \underline{P}^*((m-1).Ti) \end{aligned}\right\} i = 1 \text{ to } k,\ m \geqslant 1 \tag{12}$$

assuming that each higher repair level interval Ti is a multiple Mi of the next lowest repair level interval Ti-1. Putting i = k into (12) then gives the relationship of the state probabilities at the start of each time period Tk to the state probabilities at the start of the previous period of like duration. Equations (12) can be built up theoretically and depending on the nature of the matrices Q, Ri, i = 0 to k, may yield substantial simplification due to the sparseness of the matrices and particularly if any of the Ri, i = 0 to k, have simple patterns of 1's and 0's. If for instance at time Tk it is known that all faults are repaired then Rk has a top row of 1's and the rest 0's which when substituted in (12) gives the immediate result that

$$\underline{P}^*\ (m.Tk) = (1,\ 0,\ \ldots\ldots\ldots\ldots,\ 0) \quad m \geqslant 1 \tag{13}$$

which means that the probabilistic determination of the system is repeated for all such time intervals.

For a reliability analysis we may assume that the total system failure state is state number n. Assuming that this is a "trapping" state, and that Pn*((m-1).f) = 0 (system run does not start in this state) then Pn (m.f) is the probability of system failure during this run and the system reliability calculation becomes

$$\hat{r} = \left(\sum_{m=1}^{M} Pn\,(m.f)\right)/(m.f) \qquad (14)$$

where M is a specified number of runs. If it is not the case that Pn* ((m-1).f) = 0 then (14) is amended to remove the probability of starting a run with system in the failed state, the failure having occurred in a previous run, and is given by

$$\hat{r} = \left(\sum_{m=1}^{M}(Pn\,(m.f) - Pn^*\,((m-1).f))\right)/(m.f) \qquad (15)$$

and many of the terms of (15) usually cancel out to leave a form of equation (14) based on a repair interval rather than on the system run time f. If state n does not trap then (14), (15) become more complex but even so the information in the matrices Q, Ri, i = 0 to k, may point the way to the right expression.

The remaining issue is choice of M. If equation (13) holds good then the choice of M is simply Tk/f. If there is no time at which it is known that repair is guaranteed then M is less readily defined. However as m increases the solution to equations (12) reaches a steady state solution defined by

$$\underline{P}' = W^{(k)}\,(Tk).\,\underline{P}' \qquad (16)$$

The solution to (16) could be found by the eigen-value eigen-vector method and then $\underline{P}'$ is used instead of $\underline{P}^*$ (o) in (12) then the state probabilities would again be repeatable and M can be set to Tk/f.

As an example take a simple 2-box 'either or' system giving rise to 4 states. If the system is failed when both boxes are faulty then Q becomes

$$Q = \begin{pmatrix} -r1 & -r2 & 0 & 0 & 0 \\ r1 & & -r2 & 0 & 0 \\ r2 & & 0 & -r1 & 0 \\ 0 & & r2 & r1 & 0 \end{pmatrix} \qquad (17)$$

The repair policy states that at the end of each run only state 4 (system failure) is repaired. So Ro is given by

$$Ro = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad (18)$$

Substituting (17) and (18) into (3) and (7) and then substituting (7) into (3) gives the recursive relationships for the end-of-run state probabilities

$$\begin{aligned} P1(m.f) &= (P1((m-1).f) + P4((m-1).f)).e^{-(r1+r2).f} \\ P2(m.f) &= P2((m-1).f)e^{-r2f} + (P1((m-1).f) + P4((m-1).f))(e^{-r2f}-e^{-(r1+r2)f}) \\ P3(m.f) &= P3((m-1).f)e^{-r1f} + (P1((m-1).f) + P4((m-1).f))(e^{-r1f}-e^{-(r1+r2)f}) \\ P4(m.f) &= P2((m-1).f)e^{-r2f} + P3((m-1)f)(1-e^{-r1f}) \\ &\quad + (P1((m-1).f) + P4((m-1).f))(1-e^{-r1f})(1-e^{-r2f}) \end{aligned} \qquad (19)$$

The steady state solution of (19) is given by

$$\begin{aligned} P1' &= e^{-(r1+r2)f}\,(1-e^{-r1f})\,(1-e^{-r2f})/D \\ P2' &= e^{-r2f}\,(1-e^{-r1f})^2/D \\ P3' &= e^{-r1f}\,(1-e^{-r2f})^2/D \\ P4' &= (1-e^{-r1f})\,(1-e^{-r2f})\,(1-e^{-r1 + r2)f})/D \end{aligned} \qquad (20)$$

where $D = 1 - 3e^{-(r1 + r2)f} + e^{-(2\,r1 + r2)f} + e^{-(r1 + 2\,r2)f}$

The low-order approximations of (20) give for the system reliability

$$r = P4'/f \approx r1r2(r1 + r2)/(r1^2 + r1r2 + r2^2) \qquad (21)$$

which could not be obtained by the fault tree method. Alteration of the repair policy of even this simple model yields other such interesting results.

APPENDIX B. MPA EXECUTION PRINT-OUT.

```
**********  Model Information.  **********

There are   9 states

There are   2 basic rates with values  ->  0.2000E-04  0.1000E-03

  States            Multiples of basic rates
from    to             (1)     (2)
 1       2              6       0
 1       3              0       3
 2       3              1       1
 2       4              2       0
 2       5              1       0
 2       6              1       0
 2       7              0       1
 2       9              0       1
 3       7              2       0
 3       9              2       2
 4       7              1       1
 4       8              1       0
 4       9              2       2
 5       7              2       2
 5       9              2       1
 6       9              4       3
 7       9              3       2
 8       9              3       3

**********  Operating Conditions Information.  **********

There are   9 states with initial values  ->
 1.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

The number of maintenance types is    1  ->
  Maintenance type    1 at     0.0000 hours to states  ->
                    1   2   3   4   5   6   7   8   1

Run time =   100.0000 Flight time =    12.0000 Display interval =   111.0000
Time step =  0.1000 Initial cycles =    1 Ongoing cycles =    1 Factor =  1.0000

Number of system reliability information points is  0

**********  Output Specification Information.  **********

There are   3 outputs from   9 states  ->
  Output    1 contains    1 states  ->   1
  Output    2 contains    7 states  ->   2    3    4    5    6    7    8
  Output    3 contains    1 states  ->   9

Number of outputs on graphics plot is    2  ->    2    3

List, plot, result indicators are  ->  1  1  1

X-axis exponential grid  <-1   0   1> and Y-axis  <-1   0   1>
Number of increments on linear axes is  ->  5   5
Scales of 3 outputs on linear plot are  ->  1.000       0.5000E-01  0.1000E-03
Line trace patterns for 3 outputs are  ->  FFFF  FFFF  FF00

**********  Tabulated Probability Data.  **********

       Time.          Outputs.
          (1)          (2)          (3)

       0.000
          1.000       0.0000E+00   0.0000E+00
      12.000
         0.9950       0.5021E-02   0.5984E-05
      24.000
         0.9900       0.1001E-01   0.1800E-04
      36.000
         0.9850       0.1495E-01   0.2993E-04
      48.000
         0.9801       0.1986E-01   0.4178E-04
      60.000
         0.9752       0.2474E-01   0.5355E-04
      72.000
         0.9704       0.2958E-01   0.6524E-04
```

```
        84.000
           0.9655       0.3438E-01  0.7684E-04
        96.000
           0.9608       0.3914E-01  0.8837E-04

**********   System Reliability Data.   **********

System operation time is  ->  99.9999

Average system run probabilities  ->
           0.9777       0.2221E-01  0.4746E-04

Average rates of total system failure  ->
           0.8148E-01  0.1851E-02  0.3955E-05

Average despatch probabilities  ->
           0.9803       0.1974E-01  0.0000E+00

Average repair probability and rate  ->
          0.4746E-04  0.3955E-05

Average destination repair probabilities  ->
            1.000

**********   End of print-out.   **********
```

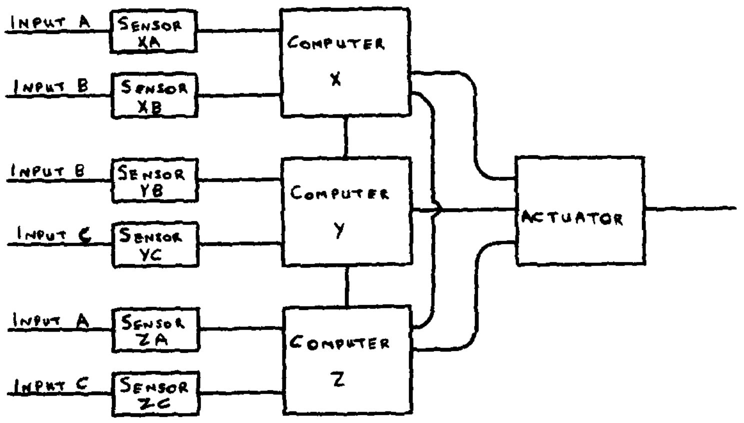


FIGURE 1. REFERENCE 2 SYSTEM. ARCHITECTURE.

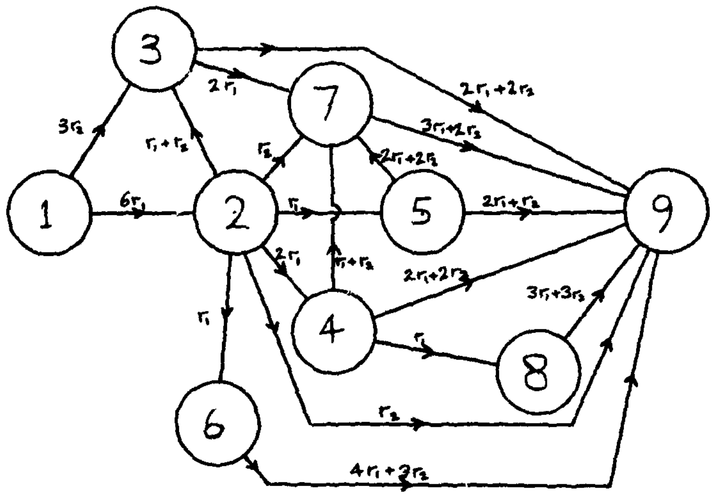


FIGURE 2. REFERENCE 2 SYSTEM. STATE TRANSITIONS.

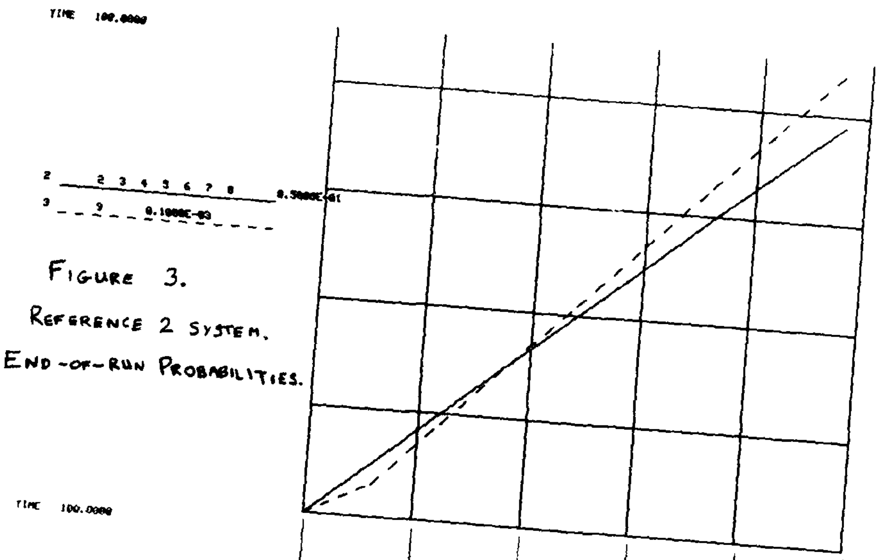


Figure 3.

Reference 2 System.

End-of-Run Probabilities.

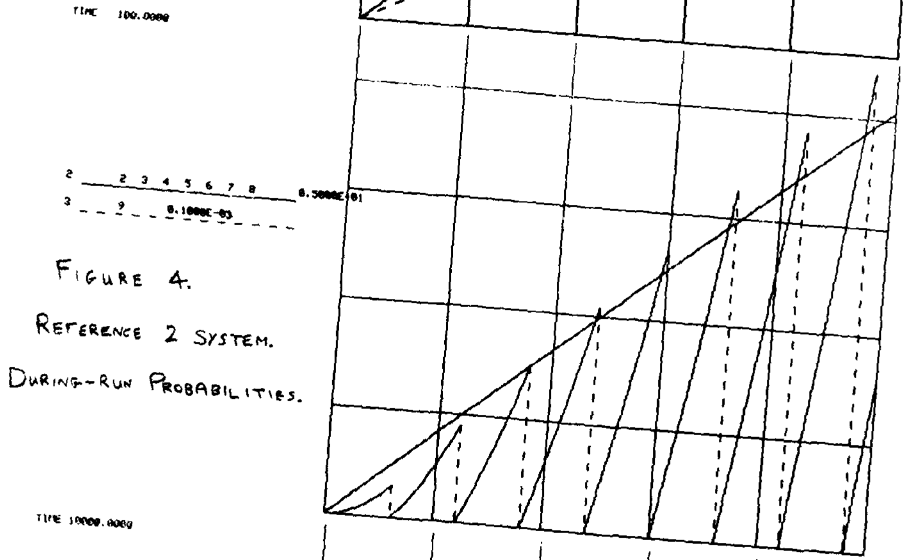


Figure 4.

Reference 2 System.

During-Run Probabilities.

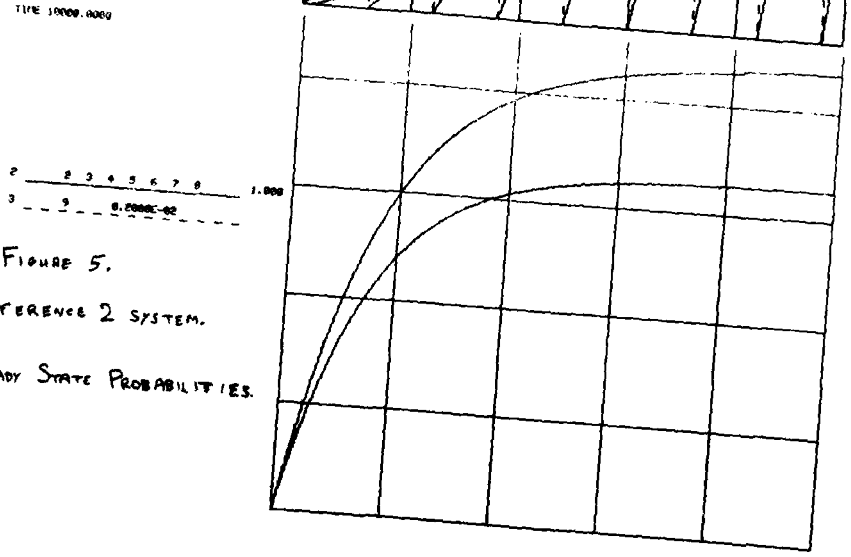


Figure 5.

Reference 2 System.

Steady State Probabilities.

TABLE 1. REFERENCE 2 SYSTEM. STATES DESCRIPTION.

State number.	State description.
1	Everything working.
2	One sensor failed.
3	One computer failed.
4	2 dissimilar sensors failed, remaining ones dissimilar.
5	2 dissimilar sensors failed, remaining ones similar.
6	2 similar sensors failed.
7	one sensor and computer failed.
8	3 dissimilar sensors failed.
9	Total system failure.

$$\begin{bmatrix}\dot{P}_1\\ \dot{P}_2\\ \dot{P}_3\\ \dot{P}_4\\ \dot{P}_5\\ \dot{P}_6\\ \dot{P}_7\\ \dot{P}_8\\ \dot{P}_9\end{bmatrix} = \begin{bmatrix} -6r_1-3r_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 6r_1 & -5r_1-3r_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 3r_2 & r_1+r_2 & -4r_1-2r_2 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 2r_1 & 0 & -4r_1-3r_2 & 0 & 0 & 0 & 0 & 0\\ 0 & r_1 & 0 & 0 & -4r_1-3r_2 & 0 & 0 & 0 & 0\\ 0 & r_1 & 0 & 0 & 0 & -4r_1-3r_2 & 0 & 0 & 0\\ 0 & r_2 & 2r_1 & r_1+r_2 & 2r_1+2r_2 & 0 & -3r_1-2r_2 & 0 & 0\\ 0 & 0 & 0 & r_1 & 0 & 0 & 0 & -3r_1-3r_2 & 0\\ 0 & r_2 & 2r_1+2r_2 & 2r_1+2r_2 & 2r_1+r_2 & 4r_1+3r_2 & 3r_1+2r_2 & 3r_1+3r_2 & 0 \end{bmatrix} \begin{bmatrix}P_1\\ P_2\\ P_3\\ P_4\\ P_5\\ P_6\\ P_7\\ P_8\\ P_9\end{bmatrix}$$

TABLE 2. REFERENCE 2 SYSTEM. MANUAL TRANSITION TABLE.

Configuration number	description.	Single failures of either a sensor or a computer. XA	XB	X	YB	YC	Y	ZA	ZC	Z
1	Everything working.	2	3	4	5	6	7	8	9	10
2	Sensor XA failed.	.	11	4	12	13	14	15	16	X
3	Sensor XB failed.	11	.	4	17	18	X	19	20	21
4	Computer X failed.	.	.	.	X	22	X	X	23	X
5	Sensor YB failed.	12	17	X	.	24	7	25	26	27
6	Sensor YC failed.	13	18	22	24	.	7	28	29	X
7	Computer Y failed.	14	X	X	.	.	.	30	X	X
8	Sensor ZA failed.	15	19	X	25	28	30	.	31	10
9	Sensor ZC failed.	16	20	23	26	29	X	31	.	10
10	Computer Z failed.	X	21	X	27	X	X	.	.	.
11	Sensors XA,XB failed.	.	.	4	X	32	X	X	33	X
12	Sensors XA,YB failed.	.	X	X	.	34	14	X	35	X
13	Sensors XA,YC failed.	.	32	22	34	.	14	X	X	X
14	Sensor XA, computer Y failed.	.	X	X	.	.	.	X	X	X
15	Sensors XA,ZA failed.	.	X	X	X	X	X	.	X	X
16	Sensors XA,ZC failed.	.	33	23	35	X	X	X	.	X
17	Sensors XB,YB failed.	X	.	X	.	X	X	X	X	X
18	Sensors XB,YC failed.	32	.	22	X	.	X	36	X	X
19	Sensors XB,ZA failed.	X	.	X	X	36	X	.	37	21
20	Sensors XB,ZC failed.	33	.	23	X	X	X	37	.	21
21	Sensor XB, computer Z failed.	X	.	X	X	X	X	.	.	.
22	Sensor YC, computer X failed.	.	.	.	X	.	X	X	X	X
23	Sensor ZC, computer X failed.	.	.	.	X	X	X	X	.	X
24	Sensors YB,YC failed.	34	X	X	.	.	7	38	X	X
25	Sensors YB,ZA failed.	X	X	X	.	38	30	.	39	27
26	Sensors YB,ZC failed.	35	X	X	.	X	X	39	.	27
27	Sensor YB, computer Z failed.	X	X	X	.	X	X	.	.	.
28	Sensors YC,ZA failed.	X	36	X	38	.	30	.	X	X
29	Sensors YC,ZC failed.	X	X	X	X	.	X	X	.	X
30	Sensor ZA, computer Y failed.	X	X	X	.	.	.	.	X	X
31	Sensors ZA,ZC failed.	X	37	X	39	X	X	.	.	10
32	Sensors XA,XB,YC failed.	.	.	22	X	.	X	X	X	X
33	Sensors XA,XB,ZC failed.	.	.	23	X	X	X	X	.	X
34	Sensors XA,YB,YC failed.	.	X	X	.	.	14	X	X	X
35	Sensors XA,YB,ZC failed.	.	X	X	.	X	X	X	.	X
36	Sensors XB,YC,ZA failed.	X	.	X	X	.	X	.	X	X
37	Sensors XB,ZA,ZC failed.	X	.	X	X	X	X	.	.	21
38	Sensors YB,YC,ZA failed.	X	X	X	.	.	30	.	X	X
39	Sensors YB,ZA,ZC failed.	X	X	X	.	X	X	.	.	27

TABLE 3. REFERENCE 2 SYSTEM. RESIDUE OF STATES AND TRANSITIONS.

State	(Configurations)	Single failure of one sensor						/ one computer.		
1	(1)	2	2	2	2	2	2	3	3	3
2	(2,3,5,6,8,9)	3	4	4	5	6	.	3	7	X
3	(4,7,10,11,24,31)	7	7	X	X	.	.	X	X	.
4	(12,16,18,19,26,28)	7	8	X	X	.	.	7	X	X
5	(13,20,25)	7	7	X	X	.	.	7	7	X
6	(15,17,29)	X	X	X	X	.	.	X	X	X
7	(14,21,22,23,27,30,32,33,34,37,38,39)	X	X	X	.	.	.	X	X	.
8	(35,36)	X	X	X	.	.	.	X	X	X

TABLE 4. REFERENCE 2 SYSTEM. BLOCKS AND ASSOCIATED DATA.

Description/Mnemonic.		Fault event number.	During-run detection.	Equivalence.	Prior-to-run detection.	Fault count.
Sensor	XA	1	X	-	No	0
Sensor	XB	2	X	XA	No	0
Computer	X	3	X	-	No	0
Sensor	YB	4	Y	XA	No	0
Sensor	YC	5	Y	XA	No	0
Computer	Y	6	Y	X	No	0
Sensor	ZA	7	Z	XA	No	0
Sensor	ZC	8	Z	XA	No	0
Computer	Z	9	Z	X	No	0

TABLE 5. REFERENCE 2 SYSTEM. PROPOGATION AND OPERATIONAL LOGIC.

Logic string of faults causing propogation.	R or I.	Blocks affected.
X	R	XA and XB
Y	R	YB and YC
Z	R	ZA and ZC

Operational logic string for model.

((XA+ZA).(XB+YB).(YC+ZC))+(XA.ZA.XB.YB)+(XA.ZA.YC.ZC)+(XB.YB.YC.ZC)

TABLE 6. REFERENCE 2 SYSTEM PLUS REPAIR. TRANSITION TABLE.

Configuration number(failed blocks).		Fault event number and mnenomic.									Repair level.		
		1 (XA)	2 (XB)	3 (X)	4 (YB)	5 (YC)	6 (Y)	7 (ZA)	8 (ZC)	9 (Z)	0	1	2
1	( )	2	3	4	5	6	7	8	9	10	.	.	.
2	(C)	.	11	4	12	13	14	15	16	XXX	.	.	1
3	(C)	11	.	4	17	18	XXX	19	20	21	.	.	1
4	(CCC)	.	.	.	XXX	22	XXX	XXX	23	XXX	.	1	1
.	.						.						
.	.						.						
.	.						.						
.	.						.						
.	.						.						
.	.						.						
39	(C CC)	XXX	XXX	XXX	.	XXX	XXX	.	.	27	.	.	1

State number.	Equivalenced fault event number. < 1 to 6 (One sensor) >						< 7 to 9 > (One computer)			Repair level. 0	1	2
1	2	2	2	2	2	2	3	3	3	.	.	.
2	4	5	5	6	7	.	3	8	XXX	.	.	1
3	8	8	XXX	XXX	.	.	XXX	XXX	.	.	1	1
4	9	9	XXX	XXX	.	.	3	XXX	XXX	.	.	1
5	9	10	XXX	XXX	.	.	8	XXX	XXX	.	.	1
6	9	9	XXX	XXX	.	.	8	8	XXX	.	.	1
7	XXX	XXX	XXX	XXX	.	.	XXX	XXX	XXX	.	.	1
8	XXX	XXX	XXX	.	.	.	XXX	XXX	.	.	1	1
9	XXX	XXX	XXX	.	.	.	8	XXX	XXX	.	.	1
10	XXX	XXX	XXX	.		.	XXX	XXX	XXX	.	.	1

TABLE 7. REFERENCE 2 SYSTEM WITH DUAL COMPUTERS. TRANSITION TABLE.

Configuration number(failed blocks).		Fault event number and mnenomic.											
		1 (XA)	2 (XB)	3 (X1)	4 (X2)	5 (YB)	6 (YC)	7 (Y1)	8 (Y2)	9 (ZA)	10 'ZC)	11 (Z1)	12 (Z2)
1	( )	2	3	4	5	6	7	8	9	10	11	12	13
2	(C)	.	14	15	16	17	18	19	20	21	22	23	24
3	(C)	14	.	25	26	27	28	29	30	31	32	33	34
4	(C)	15	25	.	35	36	37	38	39	40	41	42	43
5	(C)	16	26	35	.	44	45	46	47	48	49	50	51
6	(C)	17	27	36	44	.	52	53	54	55	56	57	58
7	(C)	18	28	37	45	52	.	59	60	61	62	63	64
8	(C)	19	29	38	46	53	59	.	65	66	67	68	69
.	.						.						
.	.						.						
.	.						.						
.	.						.						

State number.	Equivalenced fault event number.											
	< 1 to 6 (One sensor) >						< 7 to 12 (One computer) >					
1	2	2	2	2	2	2	3	3	3	3	3	3
2	4	5	5	6	7	.	8	8	9	9	10	10
3	8	8	9	9	10	10	4	11	11	11	11	.
4	12	12	XXX	XXX	.	.	13	13	13	13	.	.
5	12	14	XXX	XXX	.	.	15	15	15	15	16	16
6	12	12	XXX	XXX	.	.	17	17	17	17	18	18
7	XXX	XXX	XXX	XXX	.	.	19	19	19	19	19	19
8	4	15	17	19	16	.	4	20	20	21	21	.
9	13	15	17	19	16	.	12	20	20	22	22	.
10	13	15	15	19	18	.	21	21	22	22	XXX	.
11	20	20	21	21	22	22	13	13	23	23	.	.
12	XXX	XXX	XXX	.	.	.	24	24	24	24	.	.
13	24	24	XXX	XXX	.	.	25	25	XXX	.	.	.
14	XXX	XXX	XXX	.	.	.	26	26	26	26	26	26
15	24	26	XXX	XXX	.	.	27	27	28	28	XXX	.
16	12	26	XXX	XXX	.	.	12	27	27	27	27	.
17	12	24	XXX	XXX	.	.	12	29	29	30	30	.
18	24	24	XXX	XXX	.	.	30	30	30	30	XXX	.
19	XXX	XXX	XXX	XXX	.	.	31	31	31	31	XXX	.
20	13	27	27	29	31	.	13	24	32	32	.	.
21	13	27	28	30	31	.	13	32	32	XXX	.	.
22	25	27	28	30	31	.	24	32	32	XXX	.	.
23	32	32	32	32	32	32	25	25	25	.	.	.
24	XXX	XXX	XXX	.	.	.	33	33	XXX	.	.	.
25	33	33	XXX	XXX	.	.	XXX	XXX	.	.	.	.
26	XXX	XXX	XXX	.	.	.	34	34	34	34	XXX	.
27	24	34	XXX	XXX	.	.	24	35	35	XXX	.	.
28	33	34	XXX	XXX	.	.	35	35	XXX	XXX	.	.
29	24	24	XXX	XXX	.	.	24	24	36	36	.	.
30	24	33	XXX	XXX	.	.	24	36	36	XXX	.	.
31	XXX	XXX	XXX	XXX	.	.	37	37	XXX	XXX	.	.
32	25	35	35	36	37	.	25	33	XXX	.	.	.
33	XXX	XXX	XXX	.	.	.	XXX	XXX	.	.	.	.
34	XXX	XXX	XXX	.	.	.	38	38	XXX	XXX	.	.
35	33	38	XXX	XXX	.	.	33	XXX	XXX	.	.	.
36	33	33	XXX	XXX	.	.	33	33	XXX	.	.	.
37	XXX	XXX	XXX	XXX	.	.	XXX	XXX	XXX	.	.	.
38	XXX	XXX	XXX	.	.	.	XXX	XXX	XXX	.	.	.

TABLE 8. ENGINE CONTROL SYSTEM. BLOCKS AND ASSOCIATED DATA.

Description/Mnemonic.		Fault event number.	During-run detection.	Equivalence or rate per million hours.	Prior-to-run detection.	Fault count.
Lane A "don't care" functions.	AD	1	-	R =20.0	No	1
		2	AC+(BC.IH)	R =60.0	No	1
Lane A "accomodated" functions.	AA	3	(AC+BC).IH.BA	R =10.0	No	1
		4	AC+(BC.IH)	R =30.0	No	1
Lane A "vital" functions.	AV	5	AC+(BC.IH)	R =40.0	No	1
Lane A "multiple" functions.	AM	6	-	R = 1.0	No	1
		7	AC+(BC.IH)	R = 9.0	No	1
Lane A computer.	AC	8	AC	R =30.0	No	1
Lane A power.	AP	<	Not applicable			>
Lane A primary power.	AP/P	9	AP/P	R =100.0	No	1
Lane A secondary power.	AP/S	10	AP/S	R =70.0	No	1
Lane B "don't care" functions.	BD	11	-	E = AD(1)	No	1
		12	BC+(AC.IH)	E = AD(2)	No	1
Lane B "accomodated" functions.	BA	13	(BC+AC).IH.AA	E = AA(3)	No	1
		14	BC+(AC.IH)	E = AA(4)	No	1
Lane B "vital" functions.	BV	15	BC+(AC.IH)	E = AV(5)	No	1
Lane B "multiple" functions.	BM	16	-	E = AM(6)	No	1
		17	BC+(AC.IH)	E = AM(7)	No	1
Lane B computer.	BC	18	BC	E = AC(8)	No	1

Lane B power.	BP	<	Not applicable			>
Lane B primary power.	BP/P	19	BP/P	E =AP/P(9)	No	1
Lane B secondary power.	BP/S	20	BP/S	E =AP/S(10)	No	1
Common-mode "don't care" functions.	CD	21	AC+BC	R =20.0	No	1
Commom-mode "accomo-dated" functions.	CA	22	-	R = 1.0	No	1
		23	AC+BC	R =19.0	No	1
Single fault causes.	S	24	-	R = 3.0	No	1
Inter-lane highway.	IH	25	(AC+BC).((AD.BD)+(AA.BA)+(AV.BV))	R =17.0	No	1

TABLE 9. ENGINE CONTROL SYSTEM. PROPOGATION AND OPERATIONAL LOGIC.

1) Top level sub-model.

Logic string of faults causing propogation.	R or I.	Blocks affected.
S	I	AP and BP
CD	I	AD and BD
CA	I	AA and BA
AC	R	AA and AV
AD.AA.AV	R	AM
AM	R	AD, AA, AV
AP	R	AD,AA,AV,AM,AC
BC	R	BA and BV
BD.BA.BV	R	BM
BM	R	BD, BA, BV
BP	R	BD,BA,BV,BM,BC
AD.BD	I	CD
AA.BA	I	CA

Operational logic string for sub-model.

(AC.AA.AV)+((AC+BC).(AA+BA+AV+BV).(AV+BV).IH)+(AC.AV)+(BC.BA.BV)+(BC.BV)

2) Lane A power sub-model.

No fault propogation effects.

Operational logic string for sub-model.

P + S

3) Lane B power sub-model.

No fault propogation effects.

Operational logic string for sub-model.

P + S

TABLE 10. ENGINE CONTROL SYSTEM. FAULT COUNT, CONFIGURATIONS, STATES.

Fault Count.	Configurations.	States.	System failure rate per million hours.
1	1	1	800.00
2	21	13	89.33
3	168	68	11.87
4	706	281	7.87
5	1797	828	7.76
6	3069	1670	
7	3929	2303	
8	4265	2017	
9	4329	1395	
10	4329	439	
11	4329	439	
12	4329	439	7.74

TABLE 11. ENGINE CONTROL SYSTEM. EFFECT OF VARIATION OF REPAIR TIMES.

		Repair time for level 2 repair in hours.									
		0	100	200	300	400	500	600	700	800	
Repair time for level 1 repair in hours.	0	5.21	5.85	6.53	7.23	7.84	8.46	9.26	9.72	10.43	34.38
	50	5.50	5.86	6.56	7.25	7.87	8.50	9.32	9.78	10.48	34.75
	150	6.30	6.30	6.60	7.35	7.95	8.59	9.48	9.92	10.65	35.72
	as level 2.	5.21	5.89	6.69	7.57	8.45	9.43	10.98	11.83	13.47	47.10

Multi-Level Parallel Processing for Very High Performance Real-Time Database Management

Peter R Tillman, Ray Giles and Ian Wakley

Software Sciences Ltd
Farnborough
Hampshire
GU14 8BH
United Kingdom

SUMMARY

During the period 1980-84 Software Sciences staff, together with UK MOD research scientists, built and demonstrated a real-time distributed relational database system named ADDAM, which ran at up to 50 transactions per second on a network of 16-bit mini-computers. Since then, the company has embarked on a private venture development to scale up the throughput to over 1000 transactions per second.

This development has required the adoption of a highly parallel processing architecture. The node software has been completely redesigned so that the database can be partitioned in a single node, to enable multiple transactions to run in parallel, or one transaction to run against different parts of the database in parallel. The hardware design utilises an array of Inmos Transputers which will deliver a nominal performance of 120 mips per node.

This paper describes the overall design of the system and the benefits available from adopting such a solution to the design of military realtime systems. It also discusses the design issues which the Transputer architecture raises, the database management protocol design and the approach to maintaining database integrity in the face of concurrent parallel transactions.

INTRODUCTION

Background

Database management systems (DBMS) first made their appearance in commercial data processing in the early 1970s, with products such as IMS, ADABAS and, by 1978, IDMS. These products proposed a number of different underlying data models, IMS being based on a simple hierarchy, whilst IDMS offered a set based network.

All of these early database management systems had one very significant characteristic, which caused their databases to be very difficult to maintain and evolve over time. This was that their modelling approach required that the access paths be hard coded into the structure of the data.

By the late 1970s a new data model had emerged, the relational model [1]. This model represented a radical departure, in that it explicitly sought to avoid the encoding of access paths within the data structure. It presented a very simple data structure, 2 dimensional tables, and the design of the data tables for a given application was based on the essential characteristics of the data items without reference to the access requirements. It was this model which made distribution and parallelism a possibility.

Although by the 1980s database products had completely invaded the commercial computing world, they had made very little impact on real-time military computing. The reasons for this are two-fold; partly the way in which military computing technology developed, and partly the differences in system characteristics between the commercial and the military sector.

Development Of Military Computing Technology

In the 1960s and 1970s commercial computers were physically rather delicate, and this led to the growth of a number of specialist organisations who built rugged computers for military applications. As these machines were purpose built, they were not compatible with the existing commerical computers, and the manufacturers could not achieve the same economies of scale. These machines, therefore, tended to be very much more expensive per unit of computing power than the equivalent commercial machine. The result of this has been that, in software engineering terms, military computing has lagged a decade behind the commercial sector.

This situation is now rapidly changing. VLSI (Very Large Scale Integration) technology has brought inherent ruggedness to commercial integrated circuits. This technology can now be utilised for military applications, bringing with them the commercial economies of scale. Military computing is jumping 2 generations of hardware and this allows us to adopt commercial software engineering approaches including Database Technology.

System Characterisitics

Commercial data processing systems seek perfection in the realm of data integrity. If an error occurs in the data, the system is stopped and a recovery is run. Most commercial database products therefore provide a recovery mechanism which locks out a large part (maybe all) of the database whilst a leisurely recovery is carried out. Commercial products also tend to be transaction, rather than real-time oriented.

To apply these kind of products to a real-time military application would be totally inappropriate. Real-time systems seek high performance and availability as well as reliability. There are no commercial database products that can offer the kind of performance needed to support radar tracking or on-board avionics.

Real-Time Database Management

In 1980 Software Sciences, under sponsorship by the UK Ministry of Defence, began investigation into the problems of real-time database management. During a period of about 2 years an initial version of a system called ADDAM (Advanced Distributed Database Manager) was implemented [2, 3, 4, 5, 6 & 7]. It is not the purpose of this paper to present the work carried out on the ADDAM development. However, as it was the ADDAM work which formed the basis for the later work which this paper will discuss, it is appropriate to highlight the essential developments which came out of ADDAM.

ADDAM Fundamentals

ADDAM was conceived right from the very beginning as a distributed database system. A distributed system, properly implemented, brings two compelling benefits.

- Parallel processing
- High availability

The original version of ADDAM was implemented on 16 bit mini-computers utilising a standard Von-Neumann architecture. These computers were connected together using a bus architectured local area network. A local area network (LAN) architecture offers a true broadcast facility, which is invaluable in keeping network traffic to a minimum when employing a distributed database.

ADDAM offered a fully distributed and replicated relational database. Any number of copies of data were held around the network and each node could access all data. At the application programming level, replication and distribution were not visible. ADDAM provided reliable and synchronised propagation of updates to all nodes that held a copy of a data item affected by a transaction.

It was concluded early on in the project that the relational model was the only sound basis for a distributed database. The older types of database system tended to be heavily pointer based in their physical structure. In situations where there was a high volume of database updates, the networks costs to maintain large pointer chains would be so high, that a real-time response would be impossible. Relational tables are physically independent, their only inter-relationship is that imposed by the application.

Multi-user concurrency protection was implemented using an optimistic, no lock-on-read, protocol. Conventionally database systems protect against two users concurrently attempting to update the same record by enforcing a lock-on-read algorithm, with deadlock detection for multi-lock transactions. In a distributed database this causes a large number of messages to flow on the network. This approach also gives rise to the possibility that a longer running transaction can lock out a shorter higher priority one, because the period between lock and commit is under application control. ADDAM only locked during the execution of the commit phase. We term this an optimistic protocol because it assumes that concurrent update clashes are rare occurences. When a clash occurs the cost of rolling back the transaction that can't commit is relatively heavy.

Only one level of parallelism, that between the nodes on the network, was offered by ADDAM. This allowed a throughput of up to 50 transactions per second for a 5 node system, which was a good achievement for its time and the technology used.

THE DIOMEDES SYSTEM

The DIOMEDES system is derived from ADDAM. Whereas ADDAM was a research project, DIOMEDES is a commercial product which has taken the original ADDAM ideas and developed them very much further. The DIOMEDES hardware architecture provides three levels of parallel processing:-

- The host and DBMS functions run on physically separate processors.
- The database is distributed over multiple nodes on a network, which allows database functions on different nodes to run in parallel.
- The DBMS software within a node is arranged over an array of Transputers, which allows DBMS functions running on each Transputer to execute in parallel.

The software design also provides concurrency within a Transputer by exploiting the fact that, whilst data is being transferred over the synchronous Transputer links, the Transputer processor can carry out other work.

The incorporation of parallelism and distribution within the DBMS architecture involves a cost, namely the introduction of complex software protocols to achieve distributed data update propagation, distributed concurrency control, and resilience and recovery from node failures. Conventional systems attempt to solve these problems with the use of master/slave and handshake protocols. These protocols are costly in message passing and difficult to implement and test. DIOMEDES exploits a reliable Local Area Network System, and it has also further developed the optimistic protocol idea by introducing a buffering approach which allows for reliable update propagation without handshaking. An efficient two-phase commit protocol has also been introduced to support multi-update transactions.

HARDWARE MULTI-LEVEL PARALLELISM

Host and DBMS Separation

DIOMEDES puts the application and DBMS functions on physically separate processors (see figure 1). This separation provides many advantages.

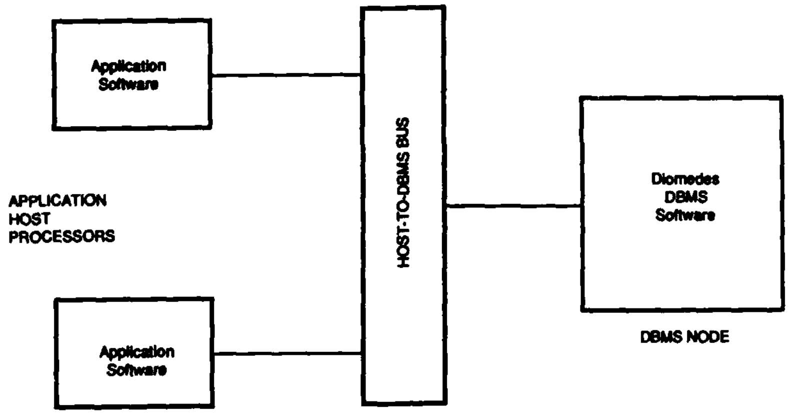


Figure 1 - Host and DBMS Separation

Whilst the DBMS is processing an application query, the host applications processor is free to carry out other applications tasks. In a general time-sharing operating system environment, this means that when one user process has issued a database request to DIOMEDES, it is descheduled to an awaiting Input/Output (I/O) state so that the host CPU can then be switched to processing another user application.

The separation of host application and DBMS functions also means that a faster database access response can be achieved. This is because the DIOMEDES hardware and software architecture has been designed and optimised to process database accesses in real-time. The DIOMEDES system contains no unnecessary operating system scheduling, I/O drivers or multi-user environment.

The DBMS to Host interface is designed to provide optimum response, as shown in figure 2.

HOST		DBMS
HOST SENDS COMMAND	---->	DBMS BEGINS EXECUTION IMMEDIATELY
	. . .	
HOST INSTRUCTS DBMS TO DEPOSIT REPLY IN THE BUFFER PROVIDED	----> <----	1ST TUPLE RETURNED, OR MORE IF THEY HAVE BEEN ACCUMULATED.
		DBMS CONTINUES EXECUTION
	. . .	
HOST INSTRUCTS DBMS TO DEPOSIT REPLY IN THE BUFFER PROVIDED	----> <----	DBMS RETURNS THOSE TUPLES WHICH IT HAS ACCUMULATED IN THE TIME THE HOST TOOK TO PROCESS THE 1ST BUFFER ETC.

Figure 2 - Host to DBMS Transaction Protocol

When a host process requests tuples, the DBMS buffer mechanisms ensure that any available tuples will be returned immediately. The DBMS does not wait until a full buffer of tuples is available before passing results back to the host. The first result buffer usually contains a single tuple so that the application obtains a result in the shortest possible time. Subsequent buffers contain multiple tuples, accumulated by the DBMS subsequent to the previous fetch, to the limit of the supplied buffer size.

Parallel Processing Provided By Distributed DBMS

A DIOMEDES system consists of a number of co-operating DBMS nodes, which communicate with each other via a Local Area Network (LAN), see figure 3. The distribution of the database over multiple nodes allows applications on different nodes to access the database in parallel.

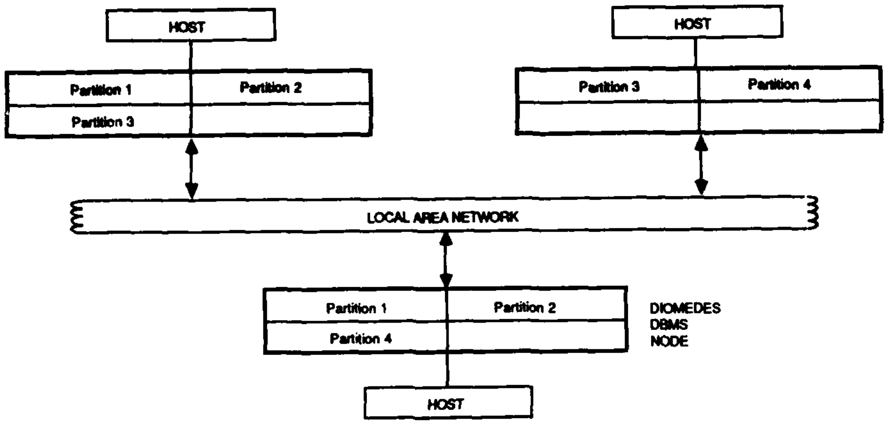


Figure 3 - Distributed, Replicated and Partitioned Database

The database may be fully replicated in all DBMS nodes, or parts of the database may be replicated on *specific nodes* only (see figure 3). This allows the parts of the database which are always accessed by the applications on a particular host, to be replicated locally to ensure optimal performance.

Within the DBMS, the unit for distribution and replication is the <u>partition</u>. Each relation may be stored in one or more partitions. Tuples of the relation are allocated to a particular partition according to the value of an attribute (or set of attributes) known as the <u>partition key</u>. A relation and the partitions in which it is stored, are illustrated in figure 4. This figure also illustrates <u>segments</u> *which are groups of attributes that are normally updated together.* As is described later, they are the unit of concurrent-update control.

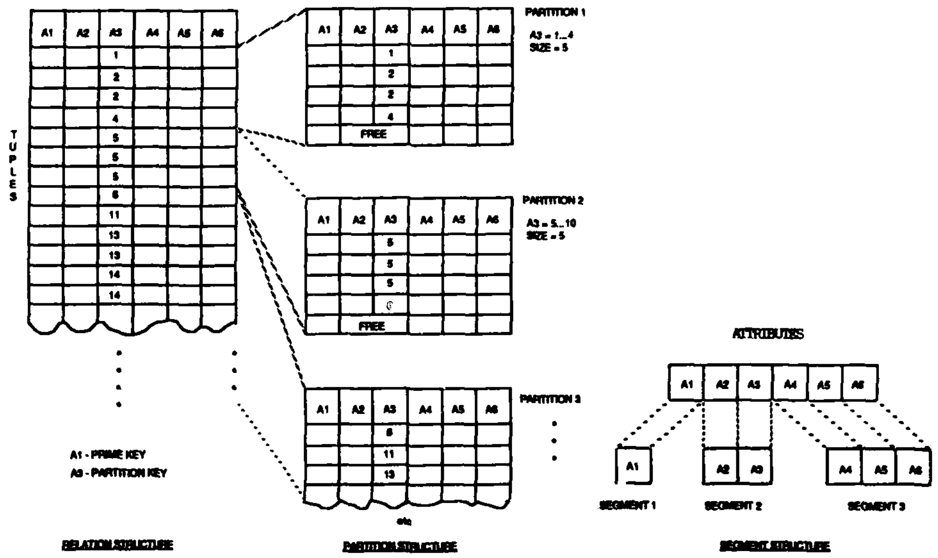


Figure 4 - DIOMEDES Data Structures

The DBMS will ensure that an update *to a replicated partition is consistently applied to all copies* in the system. The database partitions form self-contained data structures, containing no access path (pointer) information, and so an update to a tuple within a partition only affects that partition.

The database may be further partitioned or replicated within a single DBMS node to provide parallel processing at the node level, as discussed below.

Parallel Processing Within A Node

Each DIOMEDES node contains an array of INMOS Transputers. The Transputer is a single chip micro-computer containing a very fast processor (10 mips), memory and four communication links which provide point-to-point synchronous communication to other Transputers.

The Transputer can be used in networks (connected via the point-to-point links) to build up high performance concurrent systems of arbitrary size and topology. Point-to-point communication is used instead of conventional multi-processor bus architectures as it avoids communication bottlenecks regardless of the number of Transputers in the system.

Successful exploitation of the Transputer architecture requires the breaking down of the problem into a set of parallel processing tasks. For example, a search of a relation can be decomposed into a set of parallel partition searches. The tasks must then be mapped onto a topology which provides the optimium compute-to-communication ratio.

The DIOMEDES Transputer architecture is shown in figure 5. The data in each DIOMEDES node is divided into several different partition groups each of which contains a cluster of database partitions. Partitions may be replicated in a single database machine in different partition groups to reduce access contention. Each partition group is mapped onto one Transputer, which enables any given search or concurrent group of searches to be performed in parallel.

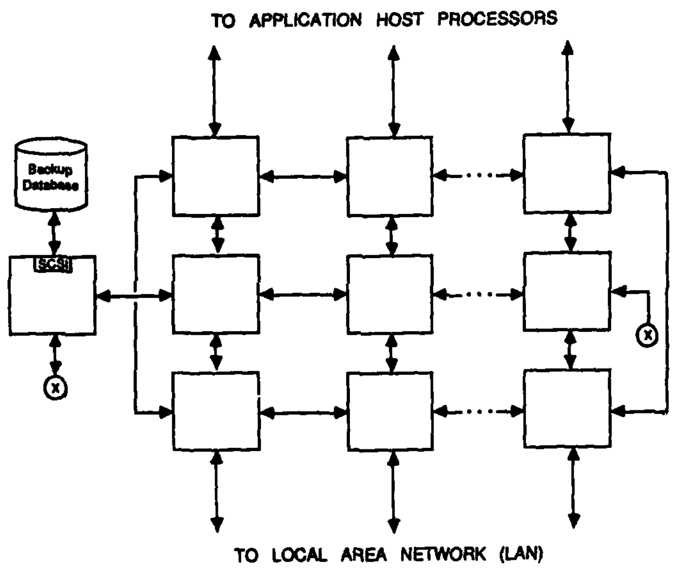


Figure 5 - DIOMEDES Transputer Architecture

Application read requests are broken down into a set of work-packages, each of which relates to a simple function (e.g. an index search of a partition). These work-packages are routed to the appropriate Transputer which is maintaining a copy of the relevant partitions.

Application update requests are broken down into a set of work-packges, each of which updates a single partition. These work-packages are routed to the appropriate Transputer for distribution over the Local Area Netowrk.

In order to achieve real-time (sub 10 milli-second) responses to application requests, all data is maintained within main memory. At regular intervals copies of the data are backed up to disc. If a partition is replicated within a node, then only one copy is backed up. The backup copy enables the system to recover the database after failures.

The software processes on each Transputer are designed to service multiple user requests concurrently (see figure 6). The software processes automatically schedule themselves to the request which most urgently requires processing; namely that where the host is awaiting further tuples or has an update outstanding.

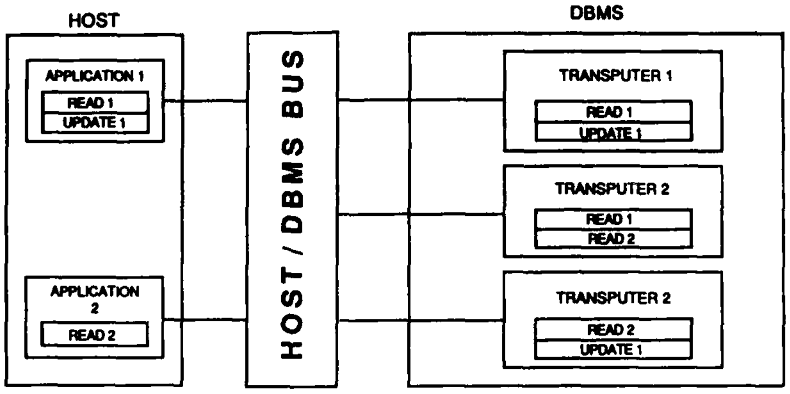


Figure 6 - Multi-Threaded Parallel Processing

<u>Decoupler And Central Servicing Processes</u>

A process on one Transputer can only send a message to a process on another Transputer if the receiver is waiting, because Transputers communicate over point-to-point synchronous links. In order to achieve a large degree of parallel processing, the processes on each Transputer have to be decoupled from each other.

The DIOMEDES design decouples the transfer of data between Transputers from the mainstream processing by interposing another process, see figure 7.

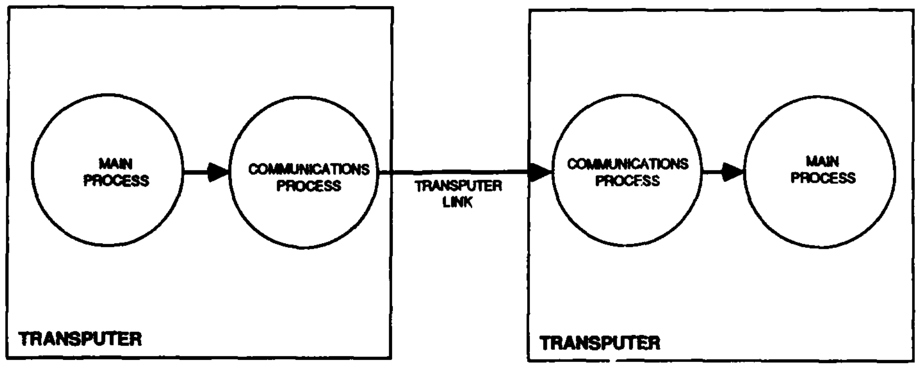


Figure 7 - Decoupled Processing and Communications

NETWORK PROTOCOLS

Reliability Protocols

The LAN ensures that messages are reliably delivered to destination nodes. As the network is asynchronous, it is possible that messages could be delivered to a node faster than that node can process them. Each node is therefore required to provide a pool of buffers large enough to accommodate the greatest peaks in message delivery that it is expected to experience. This is the responsibility of the DBMS designers for a particular application. If the node does experience buffer exhaustion then it is shut-down and re-started as though it had suffered a hardware failure.

This optimistic buffering protocol removes the need for handshaking but it does impose a higher overhead should buffer overflow occur. In addition to reducing network load, optimistic buffering reduces transaction latency by removing the need to wait for responses.

Recovery and Replication

A DIOMEDES system is loosely coupled. The failure of any single database machine connected to the network affects only the locally attached application processes. Reconnection and recovery of failed or additional nodes is achieved without breaks in system service. This allows real-time applications to be relocated and co-ordinated across any number of nodes within a distributed processing network. Applications may be co-ordinated via the database.

Each DBMS node implements its DBMS functions asynchronously from other nodes. No two nodes are dedicated in a dialogue when performing any database operation. All DBMS nodes are kept in-step by the update sychronisation and naturally "mirror" one another. Failure of a node does not require any remaining node to take specific recovery actions.

CONCURRENCY CONTROL

DIOMEDES provides concurrency control at the segment level, see figure 4. Control over such small units means that concurrent updates are very unlikely to clash. A protocol is used which is optimised for the case when no concurrency clash occurs.

The DBMS does not lock data in the conventional sense. Data may be read at any time. Updates may be attempted at any time and will only fail if the data being updated has been altered by another application, since the updating application first read it. In essence the approach is to lock-on-write rather than lock-on-read. That is, a write to a tuple locks out any prior readers from updating. When an update transaction fails due to prior update, then the whole transaction is rolled back and reported to the initiating application which must then re-start it.

This approach has three major advantages:

- The lack of read locks reduces communications traffic and transaction latency.
- Deadlock detection algorithms are not required (as they are with lock-on-read).
- Slow running transactions cannot lock out urgent ones. This is very important in real-time and military applications.

TWO PHASE COMMIT PROTOCOL

In any serious application the database management system must provide a facility for applications to issue update transactions which change more than one physical record in the database. This is termed a "Multi-Update Commit-Unit". Whilst a multi-update commit-unit is being executed, the database is inconsistent. It is consistent before the commit-unit starts and on completion of the commit-unit. The DBMS must be able to guarantee that a commit-unit once started, either completes normally or is rolled back. In a distributed database, the protocol required to achieve this is called a two-phase commit protocol.

The DIOMEDES two phase commit protocol has the following stages:

a) The initiating node broadcasts the fragments of the transaction. A fragment is an update to a segment - the unit of concurrency control.

b) All nodes receive the fragments, and those holding a copy of the data to be updated store them.

c) When all the fragments of the transaction have been broadcast the initiator broadcasts a 'prepare-to-commit' message.

d) On receipt of the prepare-to-commit, each node which has stored the fragments checks to see whether they can be applied. If a concurrency clash has occured, a 'rollback' message is broadcast.

e) If it is possible to apply the update a 'ready-to-commit' state is set to indicate a transaction is in progress on this data.

f) A 'ready-to-commit' message is broadcast indicating which parts of the transaction this node can commit.

g) When sufficient 'ready-to-commit' messages have been received, the transaction is committed.

h) If a rollback message is received, any 'ready-to-commit' flags set are cleared and the transaction fragments are discarded, so abandoning the transaction.

i) Each node will wait for sufficient 'read-to-commit' messages for a predetermined time. If a message is not received for each partition participating in the transaction (i.e. a node has failed and no copy of a segment exists) then the transaction is rolled back.

OTHER REAL-TIME FEATURES

Off-Line Query Optimization

Applications access the database using embedded query language statements. Before these can be run, the application must be processed by the DIOMEDES pre-complier, which converts the embedded query language into calls to the DIOMEDES run-time library procedures.

The DIOMEDES precompiler considers possible access path strategies and generates the most efficient query plan, ensuring that the minimum amount of processing needs to be done at runtime (see figure 8).

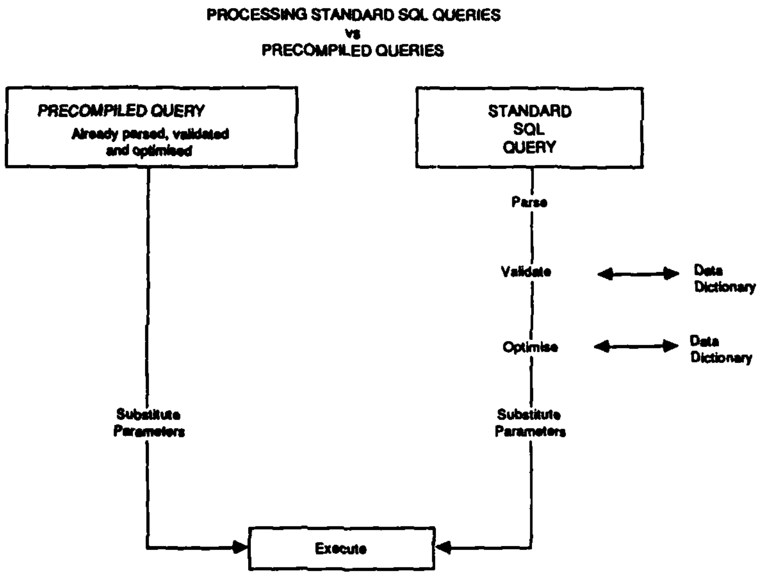


Figure 8 - DIOMEDES Precompiled Queries

The main functions of the precompiler are:

- Command verification.
- Expansion of views into base relations.
- Query optimization.
- Decompostion of the query into its basic elements, including operators to enforce referential integrity.
- Calculation of access path strategies for each query element.
- Generation of candidate query plans and selection of the most efficient.
- Generation of calls to library procedures in order to send commands to the DBMS and to fetch the results of those commands.

Event Driven Query Service

Real-time systems are characterised by the fact that they are event driven. In a DIOMEDES system, real-time events are recorded at each DIOMEDES node and are automatically propagated to all applications requiring notification of the event, see figure 9. The events are registered by updating tuples in one or more database relations. At a particular applications node, knowledge of only a subset of events may be required for the node to carry out its function. This subset may be defined by specifying the significant level of change within the relations used to store the events.

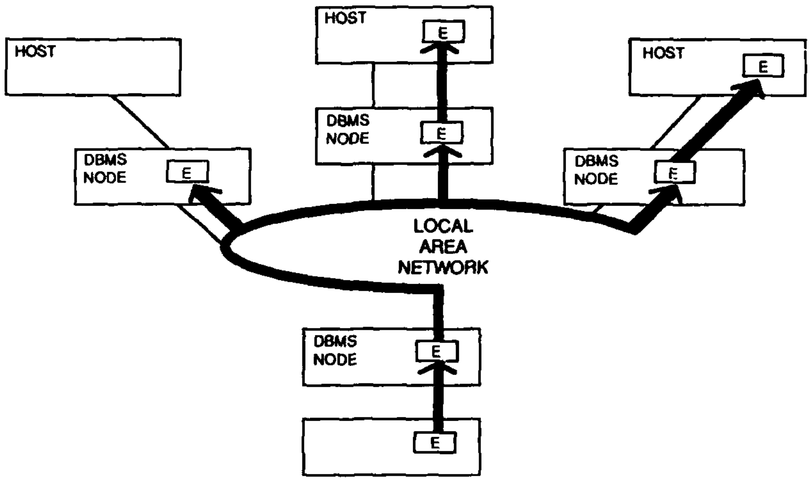


Figure 9 - DIOMEDES Real-Time Event Notification

An application can enrol to receive notification of significant events by using the MONITOR command (cf. SQL 'CURSOR') to read data from the database, see figure 10. On receipt of this command the local DIOMEDES database machine is configured to remember which database changes are significant for the particular application. Every time a "MONITORED" event occurs, the database automatically satisfies the outstanding read and returns the data to the application. This data may include not only the event-related data, but all the appropriate contextual data relevant to the event.

```
DECLARE Track MONITOR
SELECT (Trackid, range, bearing)
FROM track-table
WHERE range < 1000
AND NOTIFY
WHEN RANGE UPDATED BY 10 AND INSERTIONS AND DELETIONS;

START Track;

FETCH Track INTO no-of-track, range-of-track, bearing-of-track;

(Repeat as required)

STOP Track;
```

Figure 10 - Event Driven Monitor

Event driven query service offers four significant benefits:

- All significant events are recorded in the database.
- Each event may automatically trigger several response modules at one or more nodes in the network.
- An application processor only receives significant data from the database, and so no time is expended at the application processor in close coupled dialogues with the database. All data received is actionable.
- Any number of nodes can respond at any level of significance.

Although several responders at different nodes may action an event, each event is transmitted around the network only once.

The example above shows how a MONITOR is declared and used. The select statement defines the domain of interest, i.e. all tracks with a range of less than 1000. Initially all tracks within the domain are returned to the application. Notification is then sent of any of the following events:

- A change in the range of a track in the domain of interest by 10 or more. Note that if several changes of less than 10 occur, the application is not notified until the overall change is 10 or more since the range of that track was sent to the application.
- Insertion of a track into the domain of interest, either by insertion of a new track or by a track outside the domain having its range updated so that it is less than 1000.
- Deletion of a track from the domain of interest, either by deletion of the track, or by a track in the domain having its range updated so that it is 1000 or more.

The MONITOR start initiates the retrieval operation and sets the MONITOR position to be effectively before the first tuple of the logical table returned to the application. Fetch commands are used to copy one tuple at a time into application variables for processing. When the MONITOR is no longer required it is stopped.

SYSTEM PERFORMANCE

In order to predict the likely performance of DIOMEDES, a modelling programme was carried out as a parallel activity during the early stages of the DIOMEDES development. The team used an analytic model that employed a parameterized queuing algorithm for modelling the delays associated with Transputer processor, link and data contention. An embedded LAN model, that employed a protocol specific algorithm, was used to calculate the inter-node communications delays. All input, intermediate and modelling result data was stored within an Oracle database. This data was then interrogated by means of a Query Tool which allowed the development team to "zoom in" on the important performance problems (e.g. the most heavily loaded Transputer or the most delayed transaction) and allowed these problems to be traced back to the causal input data.

The modelling tools were recently used to predict the DIOMEDES configuration that was necessary for a system which required a performance of 450 updates and 1500 reads per second. The proposed design was a DIOMEDES system comprising 8 nodes, with each node containing 11 T414 Inmos Transputers. The database was to be fully replicated in memory at each of the nodes, and the nodes were to communicate via a 3 Mbit/sec LAN. Some 850 different accesses, each at a different frequency, were identified, to a database comprising 85 data partitions.

The specific areas of concern were in the LAN loading and the amount of parallelism that could be achieved using the Transputer architecture. The performance analysis carried out showed that the LAN loading was well within the acceptable range, but one Transputer within one of the nodes was predicted to be 88% loaded, which was judged unacceptable. This discovery led to an iteration of the design which replaced the T414 with the faster T800 floating point Transputer, and divided the heavily loaded Transputer functions over two Transputers. This resulted in a revised maximum Transputer loading of 21%. The evaluation also showed that the latency for updates and reads (sub 10 milli-seconds) were well within the requirements of the application.

APPLICATIONS OF DIOMEDES TECHNOLOGY

DIOMEDES was designed specifically to fill the gap that existed in the provision of database technology, for real-time systems. It provides a high performance and high availability relational database management system.

DIOMEDES provides both economic and technological benefits to the system implementor. It significantly reduces the complexity and cost of applications programming because it takes responsibility for all data management tasks. DIOMEDES also takes responsibility for data distribution, routing and replication management. Applications can therefore be designed and written as though they were all to run on a single centralised computer. In practice the applications can migrate around the system and the system can grow by the addition of extra nodes without any changes to application programs. DIOMEDES has been designed specifically as a distributed relational database management system. It brings benefits such as distributed two-phase commit and concurrency control which are too difficult or impractical to implement at the application level.

DIOMEDES is suitable for systems which involve hundreds of simultaneous events to which a time critical response must be made. It is capable of handling large volumes of updates and retrievals, for databases that can range up to many megabytes in size.

REFERENCES

[1] CODD, E F. A relational model of data for large shared data banks. Comm ACM 13, 6 (June 1970)

[2] TILLMAN, P R. "ADDAM - The ASWE Distributed Database Management System". Second International Symposium on Distributed Database, Gesellschaft Fur Informatik, West Berlin, September 1982.

[3] TILLMAN, P R. "A Description of ADDAM". Distributed Database in Practice, The British Computer Society, London, November 1983.

[4] REILLY M, TILLMAN P R, CATT R, MOORE R B. "Maintaining Consistency and Accommodating Network Repair in a Self-Regenerative Distributed Database Management System", 39th Symposium of the AGARD Guidance and Control Panel, Nato Advisory Group on Aerospace Research and Development, CESME Turkey, October 1984

[5] TILLMAN P R, CHEESEMAN A. "Systems Design and Data Management Problems in the Utilisation of Local Area Network Architectures", Conference on Advances in Command, Control and Communication Systems, Theory and Applications.
IEE Bournemouth, April 1985.

An updated version of this paper was also published in 1987 as chapter 3.2 in an IEE publication with the same name as the conference.

[6] TILLMAN, P R. "Maintaining Consistency and Synchronisation in Real Time Distributed Databases". MILCOMP 85 Conference. Microwave Publications, October 1985.

[7] CATT, R. "Using a Database Machine in Naval C2 Systems". MILCOMP 86 Conference. Microwave Publications 1986.

DISCUSSION

A.O.Ward, UK

We can identify the sending of data as a limitation in the performing of future airborne knowledge based systems. Do you think that your DBMS would be of use in such applications?

Author's Reply

In principle yes. The DBMS provides a parallel searching capability and this could be utilized for fast searching. There might be a problem with partitioning the data depending on the application and the nature of the knowledge based system. The DBMS could also manage the data updates, integrity, recovery, etc. and its distribution on the system bus.

W.Mansel, Ge

How have you realized synchronizations between data transfer and the data base updates?

Author's Reply

The data highway synchronizes all database updates, including those between network modes and those between strips on the same mode. The highway provides a reliable serialising transport layer.

TASK INTERACTIONS

IN DISTRIBUTED MACHINES OF EMBEDDED COMPUTER SYSTEMS

Authors: D. Flemming and Dr. H. Grundner

MBB Dynamics Division
P.O.B. 80 11 49
8000 München 80, W. Germany

SUMMARY

The programming and implementing of embedded real-time computer systems is one of the most difficult problems of data-process technology. The parallel and non deterministic actions within the systems are hard to describe and to check. Nearly all languages designed for real-time applications, as e. g. real-time FORTRAN, Coral 66, RTL/2, etc., are incomplete tools to solve the problem of concurrency because of their inherently sequential nature. The programming language Ada offers constructs for the description of task interactions in a very abstract manner without relation on the architecture of the target machine.

This paper describes the harmonization of a message driven distributed machine architecture with the programming language Ada. The problem of concurrency is described in general and a communication concept for task interactions is proposed. Analysis of the real-time requirements for future applications leads to the definition of a set of message types, where deterministic and underterministic task interactions can be achieved. Based on this definition a distributed machine architecture is presented that consists a network of computing units and an autonomous communication system, designed to reduce all administration work to a minimum. A special "no-wait-send" communication procedure is modeled in Ada language. Restrictions and augmentations for Ada programming language are worked out subsequently and demonstrated by examples. This paper shows that Ada has valuable constructs for the description of even hard real-time applications by defining the right system concept. This concept is based on a message-driven distributed machine architecture and an autonomous communication procedure.

GLOSSARY OF TERMS

APSE	Ada Programming Support Environment
ASIC	Application-Specific Integrated Circuit
C	Consumer Task
CPU	Central Processor Unit
HOL	High-Order Language
KAPSE	Kernel Ada Programming Support Environment
MB	Mail Box
MAPSE	Minimal APSE
P	Producer Task
PAMELA	Process Abstraction Method for Embedded Large Applications
PDL	Program Design/Documentation Language
RAM	Random Access Memory
VLSI	Very Large Scale Integrated Circuit

THE PROBLEM OF CONCURRENCY

Data processing under real-time conditions more and more becomes the essential key in avionic and defense systems as well as in the automation field. Real-time processing in embedded computer systems means the computing of sensor data under the conditions of undetermined events and fixed time limits for the purpose of process action and control. Beyond their future applications, as e. g. "flight control", "autonomous target identification", "multisensor fusion", "aircraft robotics", etc., they additionally request the concurrency of different tasks which correspond to each other.

However, the methods and procedures of today's data processing do not really meet the application requirements given and well-understood. This can be seen from the tremendous costs of big software projects, the instability of time-limited systems, the maintenance problem after implementation. Big efforts have been made to overcome some of the problems addressing singular aspects, such as high-order languages (HOLs), software development tools and high performance CPUs. Practice has shown that these approaches help only partially in real-time applications and that they create other problems. The problem of concurrency is generally not supported today. Five reasons will be outlined to support this statement:

- The programming method is not problem-oriented even in case a problem-oriented programming language can be used. The development costs are increasing more quickly than the complexity of the problem. The real-time problem is considered in the sequential way of programming. However, the structured analysis of the application problem creates a net of tasks interacting by message transfers. That is adequate to the problem. If conventional programming methods are used, this comprehensive network of tasks will be destroyed during implementation. This results in bad planning of real-time behavior.

o A lot of sophisticated tools for programming and execution have been invented, such as multiprocessor-multitask runtime systems, task priority management etc., to provide implementation help to the programmers. This system software helps to synchronize the process and to standardize the technical approach. However, it spoils the software and hardware transparency and so the testability. Furthermore, the useless administration efforts consume valuable processing time.

o In most computer systems the synchronization of tasks, events and hardware resources is achieved by administration in a centralized version. This fact does not make embedded systems modular, but rather very sensitive to disturbances and faults. Message-driven concepts /2/, /6/, /11/ are of great interest to solve this problem in future computer systems.

o The high-order languages (HOLs) help to establish big software packages at the cost of execution time, storage size and compilation. In real-time applications these parameters have a tremendous impact on the overall design of the target machine. Therefore, a consistent development environment is required including test facilities and real-time simulation. However, sometimes the tools become so complicated and inflexible that no reasonable tradeoff can be calculated.

o Concerning the hardware, computer buses were standardized based on administration principles that permit a computer to be configurated according to a specific application without detailed knowledge of the circuits. In multiprocessor systems of that kind bus arbitration and resource allocation have to be solved instantly by a centralized administration. In many cases the expansion of performance is therefore limited and makes the implementation very complex.

In the future, the problem of concurrency has to be solved by another approach bringing into harmony all the different requirements for standardization, regularity and flexibility of software and hardware during all phases of implementation. The resulting embedded computer systems have to be less complex and must be easy to change. The following three principles should be applied to follow new ways in the real-time data-processing world:

a) Using Ada for designing and programming real-time software packages in a standardized manner to yield all known benefits - **High Order Language (HOL).**

b) Problem-oriented software structure with direct software on hardware allocation in distributed machines - **Network Programming.**

c) Substitution of administration efforts by communication resulting in a message-driven system - **Autonomous Communication.**

A CONCEPT OF REAL-TIME COMMUNICATION

Using the known methods and procedures of structured analysis is the way of starting to solve the problem of designing real-time embedded computer systems. Real-time applications are to be described as networks of communication tasks. Application-specific network topology, computing performance, storage size, etc. are elements of description. Fig. 1 shows the general structure of embedded computer systems realized in distributed machine architecture.

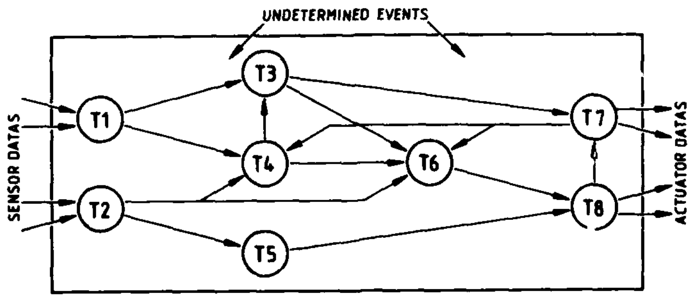


FIGURE 1: GENERAL STRUCTURE OF EMBEDDED COMPUTER SYSTEMS

The general structure requires the necessary resources to be permanently available to each task. Tast interactions are carried out by message transfers in unidirectional communication units. Various theoretical studies, e. g. /1/, /2/, /3/, describe and prove the functionability and stability of this embedded computer system realized in network structure.

In the network the messages should be transmitted in a flow-oriented and runtime-neutral way without administration efforts impeding the task runs. The communication administration necessary in the older embedded computer systems, which is a burden to the system, is now replaced by an autonomous communication system. The system has to carry out task interactions concurrently with the task actions. Only in this way can an optimum real-time behavior of the structure be achieved.

Communication concepts those can be used for the execution of task interactions are described in various publications, e. g. /4/, /5/, /6/, /7/. The usable concepts

o remote-task-send
o remote-invocation-send
o synchronization-send
o no-wait-send

will be analyzed with respect to operationability in the chapters to follow. These concepts control task interactions with different transfer procedures and software and hardware components. Fig. 2 shows principles of the communication concepts.

The communication concepts "remote-task-send" and "remote-invocation-send" are synchronous communication models. They transfer messages only on request of the consumer or producer task. In the "remote-task-send" concept task interactions are initiated by a GIVE request of the consumer task to the producer task and in the "remote-invocation-send" concept by a TAKE request of the producer task to the consumer task. These requests force the synchronization of producer task and consumer task. After synchronization was successful, the tasks exchange the message of the task interaction. In distributed architectures there are three different kinds of task interactions which are shown in table 1. Both concepts mentioned above support only task interactions of the kind: single producer to single consumer task. All further task interactions of table 1 will not be supported. However, all possibilities of realization are necessary for efficient distributed machines.

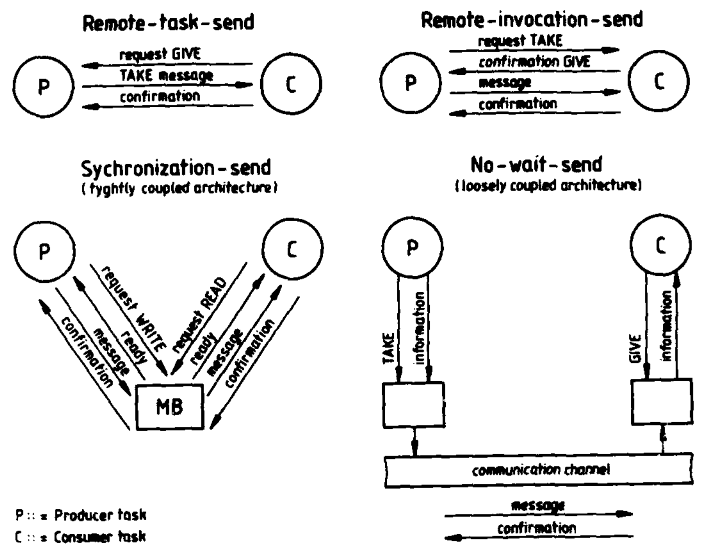


FIGURE 2: PRINCIPLES OF COMMUNICATION CONCEPTS

The communication concepts "synchronization-send" and "no-wait-send" are asynchrous communication models. Their implementations require differen‸ hardware and software structures.

The "synchronization-send" concept is designed for tightly-coupled structures. It transfers messages via multi-access structures, such as mail boxes or shared memories.

The "no-wait-send" concept is designed for loosely-coupled structures. It transfers messages via links, such as serial or parallel point-to-point lines or local area networks.

Task interactions

communication concepts	single producer single consumer	single producer multiple consumer	multiple producer single/multiple consumer
Remote-task-send	X		(X)
Remote-invocation-send	X		(X)
Synchronization-send	X	X	X
No-wait-send	X	X	X

X ::= Full support
(X) ::= Partial support

TABLE 1 : TASK INTERACTIONS SUPPORT OF COMMUNICATION CONCEPTS

The concepts "synchronization-send" and "no-wait-send" support all task interactions necessary in distributed architectures. Thus, they can be generally used to execute task interactions in distributed machines. The different forms of implementation in software and hardware are still to be examined with respect to realization efforts and transfer efficiency.

Multi-access structures, e. g. mail boxes, shared memories and similar structures, which are necessary for the "synchronization-send" concept, are passive objects of communication and require great administration efforts from the software. The accesses of producer and consumer tasks to the passive objects have to be handled by synchronization aids of the tasks or special software, e. g. semaphores or monitors. The administration effort necessary is complex and impedes the real-time behavior of the tasks. Furthermore, it cannot be reduced by efficient forms of implementation. However, the links of the "no-wait-send" concept, in various forms of implementation, can be realized as passive or active objects of communication in distributed architectures. In case they were realized as active objects, this permits the design of an autonomous communication system. The communication system has to carry out the following autonomous functions:

a) Message Setup

The producer task delivers the message to the link of its node, initiates it and then continues the task run (no-wait-send). Further messages can be delivered independently of the message transfer. Number and size are limited by the physical realization.

b) Message Transmission

The link transmits the message in a name-oriented manner via an unidirectional communication channel switched as a point-to-point line or local area network.

c) Message Receipt

All receivers connected to the link detect (filter) the message name (entry name) and - if they accept the message - receive it for the consumer tasks of their nodes.

d) Message Delivery

The receivers check the messages received and handle them independently of the consumer tasks. The messages can be processed according to type and then, depending on the type, delivered directly or on request to the consumer tasks of the nodes.

The "no-wait-send" communication concept is the most effective one of the four concepts examined. The functional structure of the concept depicted above is data-flow-oriented and represents a complete autonomous communication system. This guarantees a minimalization of the administration efforts on the task and software side. This is the case because the software effort for task interactions is reduced to mere deliver and receive actions of the producer and consumer tasks.

In order to guarantee run conditions defined within the distributed architecture and in order to achieve the necessary flexibility for programming, the types of messages for real-time applications have to be defined in the following. They have to be compatible with the syntax of the Ada reference manual /8/. Furthermore, experience gained from real-time applications and examinations of them require the real-time messages to be classified into types.

In the suggested distributed machine architecture of embedded computer systems a set of messages structured in data messages and system messages is defined for task interactions. Table 2 shows the defined set of messages. All types of messages are transmitted via the autonomous communication system. Fig. 3 shows the information formats of the messages to be transmitted.

Messages

Data messages	System messages
Type: State data	Type: Synchronization promise
Event data	Exception handling request
Expiration data	Abort request

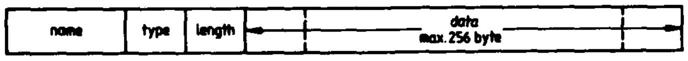
TABLE 2: TYPES OF MESSAGES

Both information formats of messages start with the name of the message. The name refers to names of entry declarations in Ada programs. The name helps the receivers to detect (filter) the messages addressed to the consumer tasks of their nodes.

System messages contain the name of the current action only. The receivers transfer the names of system messages received directly to the consumer tasks of their nodes. After receipt, the tasks start the program actions to be allocated to the names of system messages.

Data messages transfer parameters of the Ada entry calls and, in addition to the name representing a task entry in consumer tasks, contain the data and details about the type and the length of the data. The latter information serves for controlling purposes for the receivers of the communication system. The type of data, however, is an administrative information for the receivers. It determines how the data received are to be treated. Data messages transfer parameters of the Ada entry calls as "state data", "event data" or "expiration data" in the distributed architecture.

DATA MESSAGE:



name ::= [task_name.] entry_name
type ::= state | event | expiration
length ::= number_of_data_bytes

SYSTEM MESSAGE:

name ::= task_name | system_message
system_message ::= [task_name.] system_message_identifier
system_message_identifier ::= entry_name | exception_name

FIGURE 3: INFORMATIONFORMATS OF MESSAGES

"State data" are state values of the embedded computer system or its environment related to a time interval. This time interval is determined by the cycle time of the producer task which updates the data. Consumer tasks can access to the values of the "state data" as often as they wish. Old values of "state data" stored in the receivers are overwritten by new values to "state data" (same name) of the producer tasks.

An example of a typical "state data" message is the time often required in real-time applications. In the distributed architecture a producer task centrally updates the time every second and sends the updated values of the parameters SEC, MIN, HOUR, DAY, etc. as "state data" messages to the receivers of the consumer tasks. They can interrogate the time from the receivers of their nodes as often as they wish, as they are handled as "state data".

"Event data" report about certain (single) values, as e. g. about a certain measured value or a certain character. The values of "event data" messages store receivers for single accesses of the consumer tasks. After a read access by a consumer task to a current value of "event data", the value in the receiver of the task's node becomes invalid and is no more available to further read accesses. Upon read accesses, the stored follow-up values of "event data" (same name) deliver receivers to consumer tasks of their nodes in first-in-first-out order. The values of "event data" stored in the receivers are only overwritten by new values to "event data" (same name) of the producer tasks, if they became invalid after read accesses.

Typical "event data" messages are, e. g., parts of ordered data streams, such as sequences of characters, which supply the correct results only if they are executed or evaluated in the correct order. Values of messages agreed as "event data" buffer and transfer receivers to the consumer tasks of their nodes in the correct order. Thus the receivers handle the data buffers of the "event data" messages for the consumer tasks.

"Expiration data" are values of the embedded computer system or its environment valid only for a limited period of time. Within their lifetime, the values of "expiration data" can be accessed to by consumer tasks as often as they wish. The receivers autonomously control the lifetime of the values. It can be set and may amount to a multiple of a basic time clock. This handling is limited by the physical realization. Valid and invalid values of "expiration data" stored in the receivers are overwritten by new values to "expiration data" (same name) of the producer tasks. By doing so the lifetime of the values is initialized again in the receivers.

Values that can only be correctly executed or evaluated within their lifetime, which is determined by signals allocated to the values, are typical "expiration data" messages.

The distributed machine described in /11/ uses a similar concept to transfer and handle the messages of task interactions.

THE ARCHITECTURE OF DISTRIBUTED MACHINES

The suggested distributed machine consists of a cluster of subsystems cooperating with each other and realized in VLSI-technology and a communication system connecting the subsystems with each other in a flow-oriented way. Fig. 4 shows the general structure of this cluster.

A subsystem is a hardware and software unit carrying out a complete task in parallel to the tasks of other subsystems of the cluster. A subsystem consists of a processor node including communication units and the total software the task needs to carry out its activities.

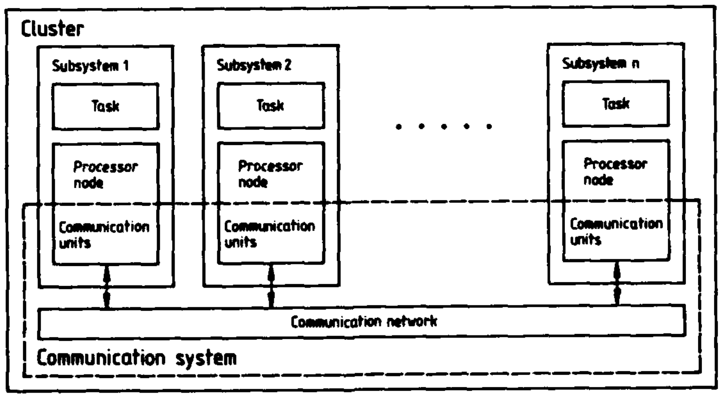


Figure 4 : CLUSTER

In the suggested distributed architecture a processor node is a hardware unit consisting of at least the VLSI components microcomputer (processor with memory) and communication unit. Depending on application the processor node can be expanded by further communication units and a process interface, which - if required - permits sensors and actuators to be coupled to processor nodes to measure and/or generate signals. Fig. 5 shows the general structure of a processor node.

The communication system of the architecture can be regarded as a network of loosely coupled communication units and links which - in a flow-oriented way - are switched as unidirectional point-to-point lines and/or local area networks. It consists of the following assemblies:

o Communication units of the processor nodes. The structure of a communication unit is shown in fig. 6.

o Communication channels.

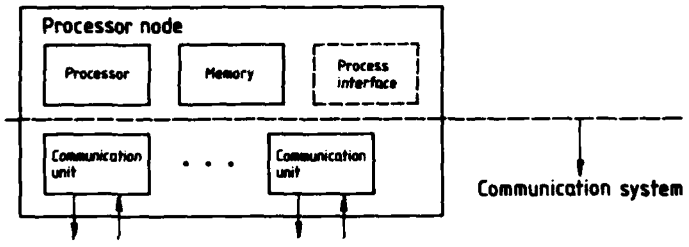


FIGURE 5: PROCESSOR NODE

The communication unit consists of a transmitter unit and a receiver unit. Both units are coupled to a processor node and work independently of each other.

The producer task has to inform the transmitter unit about the name, type and length of each message in order to be able to transmit the messages of task interactions autonomously. Upon receipt of this information the transmitter unit starts the autonomous message transfer of the communication channel connected.

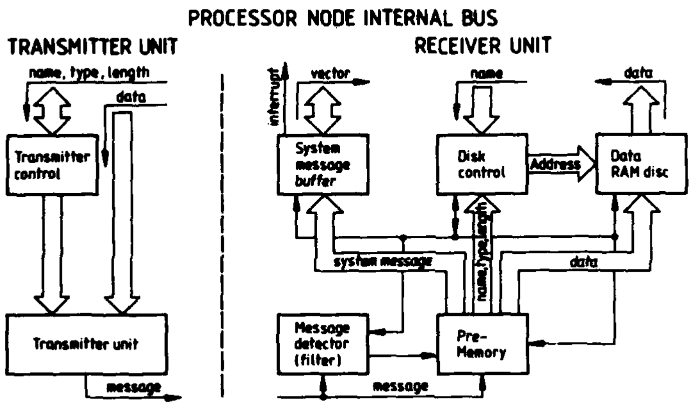


FIGURE 6: COMMUNICATION UNIT

The program has to inform the receiver unit about all names of the messages which can be transmitted to the consumer task of its processor node in order to be able to receive messages of the task interactions autonomously. With the help of the list of names the receiver unit detects (filters) the messages of the communication channel connected. It accepts messages with names from the communication channel which the consumer task of its processor node accepts as well. The receiver unit transmits system messages accepted directly to the consumer task via the system message buffer. Data messages, however, are stored and processed by the receiver unit in the data RAM disk for read accesses of the consumer task.

The communication network is switched in a flow-oriented way. This means, the suggested distributed machine architecture does not execute task interactions via universal, system-oriented communication channels, but rather via application-oriented channels switched.

The following links are available for the realization of distributed machine architectures; they can be designed with the help of Ada language constructs, as shown in the chapter "Communication Model in Ada", and have been realized in the form of:

o Private communication channels, point-to-point lines between a transmitter and a receiver;

o global communication channels, local area networks between one or several transmitters and one or several receivers.

Fig. 7 shows forms of the unidirectional communication channels permitted in the suggested distributed architecture.

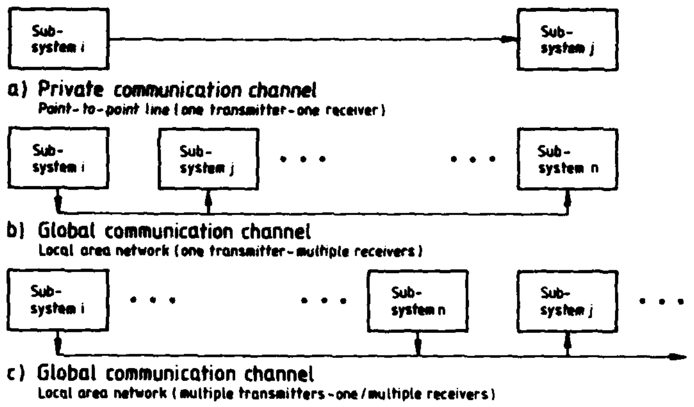

Sub-system i
Sub-system j
a) Private communication channel
Point-to-point line (one transmitter-one receiver)
Sub-system i
Sub-system j
Sub-system n
b) Global communication channel
Local area network (one transmitter-multiple receivers)
Sub-system i
Sub-system n
Sub-system j
c) Global communication channel
Local area network (multiple transmitters-one/multiple receivers)

FIGURE 7: COMMUNICATION CHANNELS

Private communication channels transmit messages between a producer and a consumer task. The point-to-point lines of the channels directly correspond to task interactions, the actions of which in Ada programs represent an entry of a consumer task and an entry call of a producer task, each of the same name.

Global communication channels transmit messages between several tasks.

o Being occupied by a transmitter and several receivers, global communication channels transmit messages between a producer task and several consumer tasks. The local area networks of the channels directly correspond to task interactions, the actions of which in Ada programs represent entries (same name) of several consumer tasks and an entry call of a producer task affecting these entries.

o Being occupied by several transmitters and one or several receivers, global communication channels transmit messages independent of each other of several producer tasks to one or several consumer tasks. The local area networks of the channels directly correspond to task interactions, the actions of which in Ada programms represent entry families (same name) of one or several consumer tasks and the entry calls of several producer tasks affecting the distinct entries of the families.

COMMUNICATION MODEL IN ADA

The task interactions to be executed in the suggested distributed machine architecture are be designed and documented as transparent software and hardware allocations in Ada programs. For this purpose the syntax and semantics for tasks and task interactions determined by Ada is to be checked and, if necessary, to be adapted to the operating conditions of the suggested distributed machine architecture.

Ada /8/, /9/, /10/ defines tasks as concurrent program units working on their own "logic processors" independently of other tasks, except for points of synchronization. This ideal definition for the implementation of the suggested distributed machine is restricted by the "parent-child" dependence of the concurrent progam units, e. g. between the main program (master) and all tasks or between one parent task and its children tasks. The semantics of the task dependence requires that all tasks either be started after the master and children tasks after their parent tasks or that they be terminated before them. It is to be examined whether this task dependence which is useful in single processor systems can be applied in an analog way by the distributed architecture.

Data message transfers of task interactions are controlled as synchronous communication procedures by Ada by methods of "rendezvous technology". Fig. 8 depicts the language constructs to describe a rendezvous. In Ada tasks execute data transfers of taks interactions via an entry call of the producer task and an accept statement of the consumer task to an entry of the same name. After execution of the statements contained in the accept statement of the consumer task the rendezvous is dissolved and the tasks continue their activities in parallel.

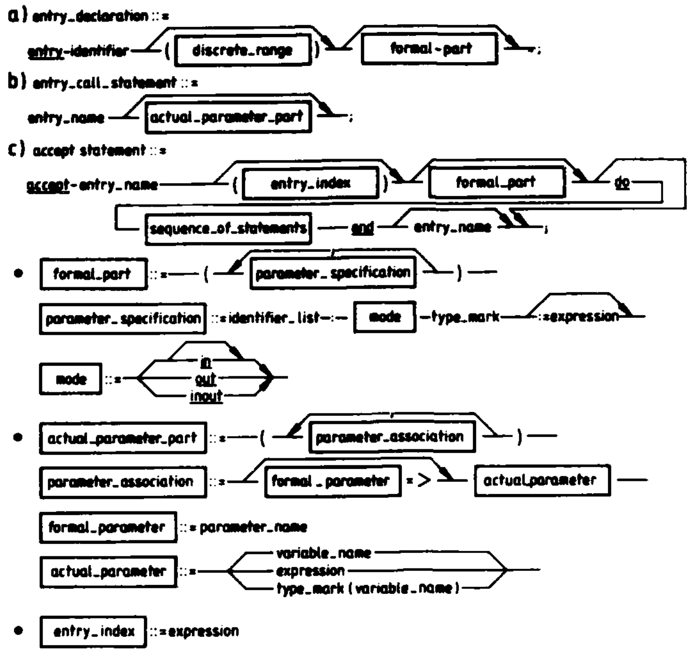


FIGURE 8: STATEMENTS OF ADA TASK INTERACTIONS

In Ada programs the data transfer of a task interaction in the producer task is noted as an entry call in the form of a subprogram call. Return parameters are permitted. The are calculated or generated within the rendezvous by the consumer task analog to the subprogram technology and then transmitted back to the producer task. The consumer task accepts the task interaction with an accept statement. This statement must calculate or generate the return parameters with the help of it's internal sequence of statements. Thus the Ada rendezvous technology also permits the compulsory sequentialization of parallel program units. Programming and control possibilities using the statements of the universal Ada rendezvous technology are to be determined for the task interactions in the suggested distributed machine architecture.

The basic form of the Ada data transfer rendezvous can be designed with the help of the language constructs outlined in fig. 8. Tasks executing an entry call or an accept statement in the basic form of the rendezvous wait for synchronization with the task taking part in the rendezvous. If there is no synchronization, the tasks will stay in a wait state and are blocked. This situation is monitored and controlled by Ada with additional selective constructs shown in fig. 9.

Selective constructs give alternatives to run for rendezvous that cannot take place. The will be executed in case producer or consumer tasks cannot be synchronized with the tasks taking part in the rendezvous. In the selective construct the "delay alternative" defines the maximum time tasks should wait for synchronization so that a rendezvous can take place. If there is no "delay alternative" in a selective construct, alternatives are started to run immediately, in case the language constructs initiating the rendezvous do not synchronize the tasks.

System message transfers of task interactions are controlled by Ada by methods of various communication procedures. Synchronization promises, like data messages, are transmitted by methods of synchronous communication procedures. Requests for exception handling and abort requests are to be noted in producer tasks by means of specific constructs. The requests are transmitted in asynchronous mode. In consumer tasks the requests initiate the immediate execution of the actions requested, as e. g. exception handling: EXCEPTION_FLIGHT_CONTROL.

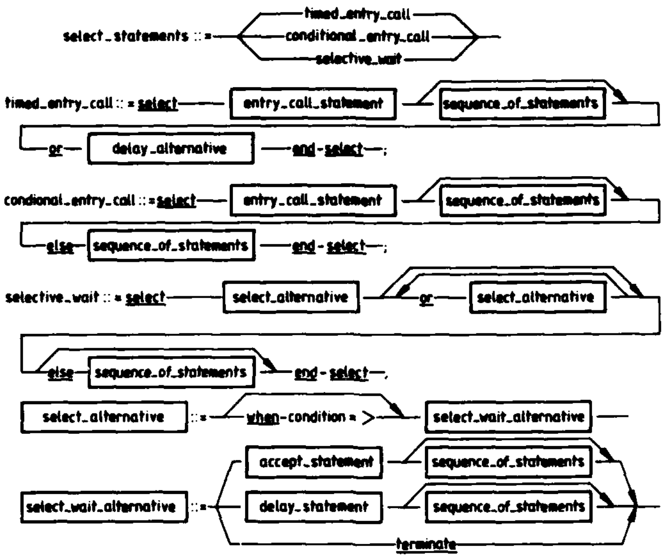


FIGURE 9 : SELECT STATEMENTS OF ADA TASK INTERACTIONS

Ada programs must be transparent and transferrable to the architecture as suggested in the chapter *"The Architecture of Distributed Machines"*. It is to be checked which restrictions and/or expansions are necessary. By doing so, the following aspects of paralleling and controlling are to be considered:

o The task dependence is given implicitly by the communication system of the architecture.

o Task interactions are not directly carried out in synchronous mode between the tasks, but rather indirectly in asynchronous *"shared rendezvous"* mode via the active communication system of the architecture.

o The communication units (transmitters or receivers) of the communication system are rendezvous partners of the producer and consumer tasks in the "shared rendezvous" mode. The units accept or deliver the messages unidirectionally.

o All task interactions are to be noted in Ada constructs taking into account the unidirectional flow in the communication channels of the architecture. Thus, the modes "out" and "in out" are not permitted in the formal part of the entry call and the accept statements. This is the only restriction required by architecture, as against /8/.

o The message types *EVENT*, *STATE* and *EXPIRATION* of the task interactions are to be specified by compiler instructions (pragmas) as instructions to the data administration of the communication system. The syntax is shown in fig. 10.

o The same entry names in tasks are necessary to design global communication paths. /8/ does not explicitly prohibit this definition. The manual does not provide a clear statement on this matter.

An analysis of the Ada syntax showed that in addition to the *transfer restriction* of task interactions to the mode "in" only the compiler instructions depicted in fig. 10 are necessary to port Ada programs to the distributed machine architecture. Transparency can be guaranteed by the autonomous communication system.

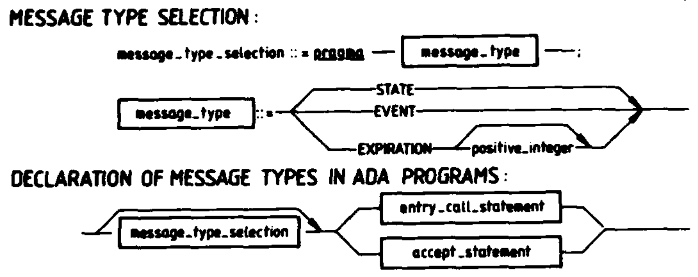


FIGURE 10: SYNTAX OF MESSAGE TYPE SELECTION

Depending on the operational requirements to be met by the embedded computer systems and in accordance with the Ada reference manual /8/ the following runtime conditions must be realized in order to be able to efficiently execute concurrent actions of tasks in the suggested distributed architecture:

o Waiting for initialization

o Activation (runtime)

o Wait for synchronization promise

o Execution of exceptions

o Termination with execution of task-specific exception handler

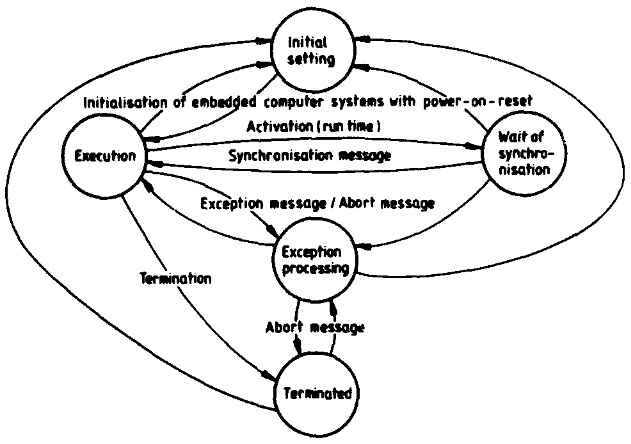


FIGURE 11: RUN-TIME CONDITIONS OF TASKS

Fig. 11 shows the runtime conditions of the tasks. All tasks start initializing the embedded computer system. As long as they are not waiting for synchronization promises, they execute their actions independently of each other. They execute task interactions as producer or consumer tasks in "shared rendezvous" mode by corresponding transmitters or receivers of the communication units of their processor nodes. They deliver the messages to transmitters and accept them from receivers of the communication system in asynchronous mode. They only get synchronized with the transmitters or receivers of the autonomous communication channel of the distributed architecture which, parallely to the task run, transmits and handles the messages. Fig. 12 shows the structure of the "shared rendezvous" mode.

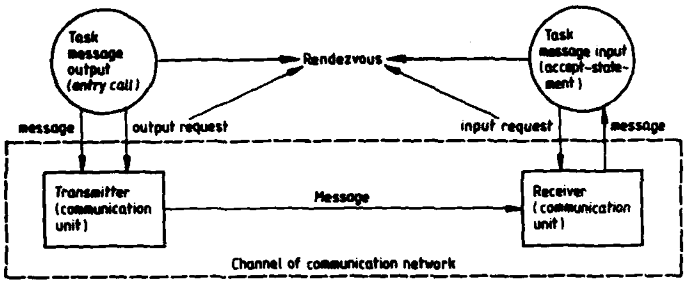

Task message output (entry call)
Rendezvous
Task message input (accept-statement)
message
output request
input request
message
Transmitter (communication unit)
Message
Receiver (communication unit)
Channel of communication network

FIGURE 12: SHARED RENDEZVOUS MODE

The application-specific communication network of the distributed architecture necessary to carry out the task interactions is describable in Ada programs. There, entry declarations represent the communication channels of the distributed architecture. The entry names of the declarations define the communication network of the task interactions, applying the following rules:

o One entry name::=
  Point-to-point lines between one consumer and one producer task (private communication channel).

o Several identical entry names::=
  Local area network between several consumer and one producer task (global communication channel).

o One or several identical entry families::=
  Local area network between one or several consumer tasks and one or several producer tasks (global communication channel).

An expansion by the attribute task name of the allocation rules outlined condenses the communication networks designed on the basis of entry names by multiple use of the communication channels, namely sequential task interactions.

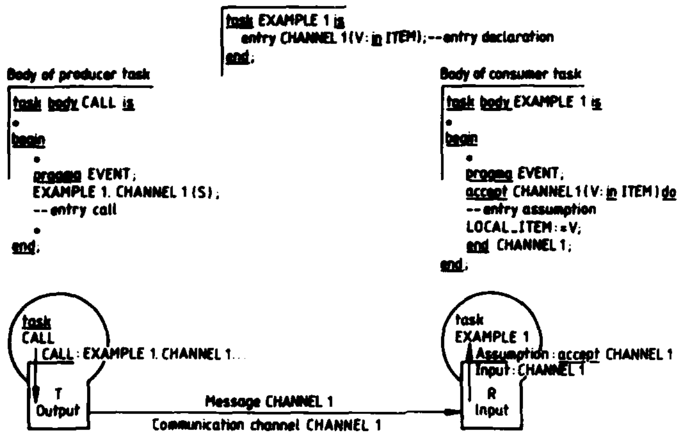
In the suggested distributed architecture the tasks get terminated independently of each other. The task dependence required by Ada handles the communication system of the architecture. It buffers the messages of producer tasks already terminated and delivers them to consumer tasks still activated.

EXAMPLES OF TASK INTERACTIONS AND DISTRIBUTED MACHINE ARCHITECTURES

The suggested distributed architecture transmits messages of task interactions via private and global communication channels with definite software and hardware allocation. This statement is confirmed by the following examples:

a) Task interactions between one producer and one consumer task:

   The message transfers are executed by the communication network having private communication channels (point-to-point lines). Fig. 13 shows an example of the software and hardware allocation of these task interactions. In this example an "event data" message is transmitted. The example also contains the necessary type declaration of the message.



```
task EXAMPLE 1 is
   entry CHANNEL 1(V: in ITEM);--entry declaration
end;
```

Body of producer task

```
task body CALL is
•
begin
   •
   pragma EVENT;
   EXAMPLE 1. CHANNEL 1(S);
   --entry call
   •
end;
```

Body of consumer task

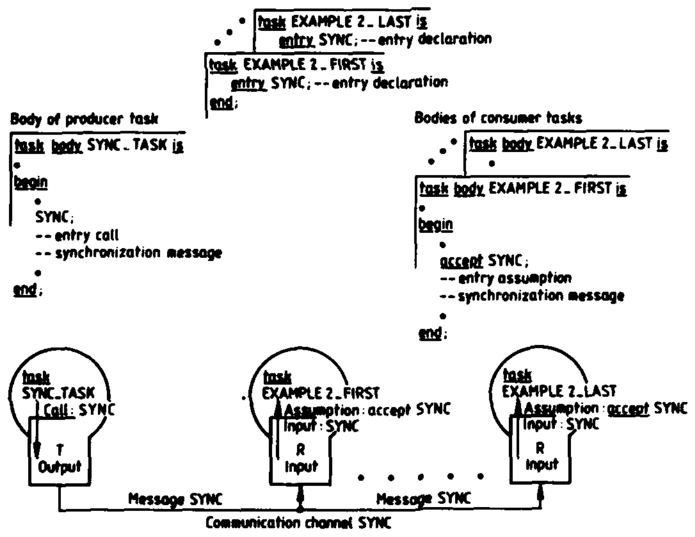
```
task body EXAMPLE 1 is
•
begin
   •
   pragma EVENT;
   accept CHANNEL 1(V: in ITEM) do
   --entry assumption
   LOCAL_ITEM:=V;
   end CHANNEL 1;
end;
```

task
CALL
CALL: EXAMPLE 1. CHANNEL 1...
T
Output
Message CHANNEL 1
Communication channel CHANNEL 1
task
EXAMPLE 1
Assumption: accept CHANNEL 1
Input: CHANNEL 1
R
Input

FIGURE 13: TASK INTERACTION BETWEEN ONE PRODUCER TASK AND ONE CONSUMER TASK

b) Task interactions between one producer task and several consumer tasks:

The message transfers are executed by the communication network having global communication channels (local area networks). Fig. 14 shows an example of the software and hardware allocation of these task interactions. In this example a "system" message (synchronization promise) is transmitted. The type of this message is already determined and does not have to be explicitly declared in the program.



```
task EXAMPLE 2_LAST is
   entry SYNC;--entry declaration

task EXAMPLE 2_FIRST is
   entry SYNC;--entry declaration
end;
```

Body of producer task

```
task body SYNC_TASK is
•
begin
   •
   SYNC;
   --entry call
   --synchronization message
   •
end;
```

Bodies of consumer tasks

```
task body EXAMPLE 2_LAST is
   •

task body EXAMPLE 2_FIRST is
•
begin
   •
   accept SYNC;
   --entry assumption
   --synchronization message
   •
end;
```

FIGURE 14: TASK INTERACTION BETWEEN ONE PRODUCER TASK AND MULTIPLE CONSUMER TASK

c) Task interactions between one or several producer tasks and one or several consumer tasks:

The message transfers are executed by the communication network having global communication channels, as shown in fig. 14. Deviating from the example shown in fig. 14, in this communication mode one or several producer tasks transmit messages of the task interactions to distinct entries of entry families of one or several consumer tasks.

d) Structure of a task network:

Fig. 15 shows the structure of a guidance & control software implemented as a distributed system according to the specifications made in this suggestion.

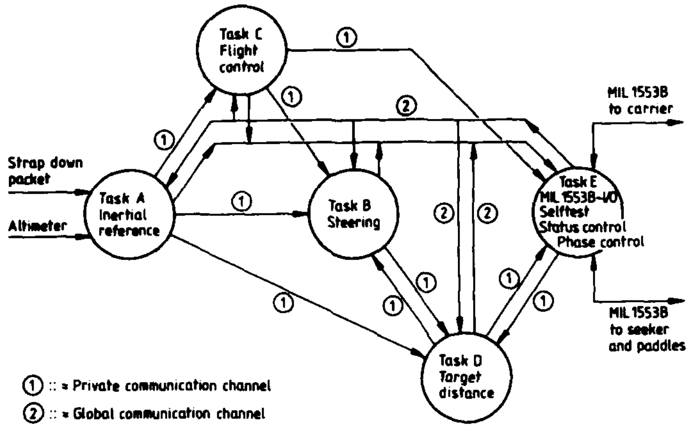

Task C Flight control
Task A Inertial reference
Task B Steering
Task D Target distance
Task E MIL 1553B-I/O Selftest Status control Phase control
Strap down packet
Altimeter
MIL 1553B to carrier
MIL 1553B to seeker and paddles
① :: = Private communication channel
② :: = Global communication channel

FIGURE 15: EXAMPLE OF MISSILE GUIDANCE & CONTROL SYSTEM REALIZED IN DISTRIBUTED MACHINE ARCHITECTURE

DESIGN AND DOCUMENTATION TOOLS

An APSE, consisting of the MAPSE tools and program abstraction tools, such as PAMELA /12/ or Buhr Design Method, and runtime analysis/communication simulator are necessary for the computer-aided support of the implementation of Ada programs on application-oriented distributed architectures. The tools must support the separation of the tasks as complete *loadable and operable program* units. They must automatically generate and check the names of the flow-oriented message transfers of task interactions in all operating levels and for all separated tasks.

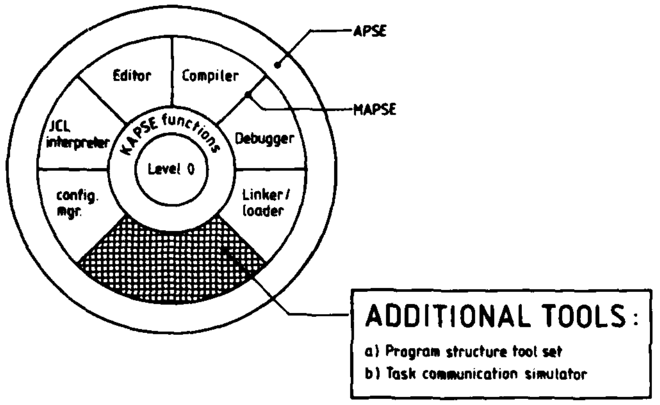


FIGURE 16: ADDITIONAL TOOLS FOR COMMUNICATION AND NETWORK DESIGN

The entry call for task interactions defined by Ada is to be accepted by the tools only unidirectionally in mode "in". As far as the handling of the data flows of task interactions by the hardware is concerned, i. e. the communication system of the suggested distributed architecture, the compiler instructions (pragmas) STATE, EVENT and EXPIRATION are to be checked and executed additionally. The shared variables permitted by Ada are generally uneffective in distributed architectures and therefore must not be supported by the tools.

CONCLUSION

Ada language constructs are a valuable implementation tool to execute task interactions in embedded computer systems realized as distributed machines according to the suggested concept of architecture. Once the semantics has been clearly defined, the Ada constructs can be used to design, program and implement application-oriented distributed architectures for real-time applications with direct hardware and software allocation. The suggestion outlined confirms this.

REFERENCES

/1/ V. Nguyen, A. Demers, D. Gries and S. Owicki: "A model and temporal proof system for networks of processes", Distributed Computing, Vol. 1, No. 1, 1986

/2/ H. V. Issendorff: "Theory of organization", Bericht-Nr. 361, Forschungsinstitut für Funk und Mathematik, Wachtberg-Werthhofen, W-Germany

/3/ B. Szymanski, Yuan Shi and N. S. Prywes: "Synchronized distributed termination", IEEE Transactions on Software Engineering, Vol. 11, No. 10, 1985

/4/ E. W. Dijkstra: "The structure of the THE-multiprogramming system", Communication of the ACM, Vol. 11, No. 5, 1968

/5/ E. W. Dijkstra: "Guarded commands, non determinacy and derivation of programs", ACM Computing Surveys, August 1975

/6/ K. J. Thurber and G. M. Masson: "Distributed processor communication architecture", Lexington Books, D. C. Heath, Lexington, Mass.

/7/ D. M. Topkis: "Concurrent broadcast for information dissemination", IEEE Transactions on Software Engineering, Vol. 11, No. 10, 1985

/8/ G. Goos and J. Hartmanis: "The programming language Ada reference manual", ANSI/MIL-STD-1815-1983, Springer-Verlag

/9/ V. A. Downes and S. J. Goldsack: "Programming embedded systems with Ada", Prentice-Hall International, Englewood Cliffs, N.Y.

/10/ G. W. Cherry: "Parallel programming in ANSI standard Ada", Reston Publishing Company, Reston, Virginia

/11/ W. Merker: "Ein wartbares, verteiltes Realzeitsystem", Regelungstechnische Praxis, Heft 6, Juni 1984

/12/ G. W. Cherry: "PAMELA designer's handbook", The Analytic Sciences Corporation

DISCUSSION

W.Mansel, Ge

To realize your "No-Wait-Serial" task interaction mechanization will probably need some mechanism which might be realized by special I/O processes. Can you comment on this?

Author's Reply

The I/O processes of producer tasks are entry-calls of mode "in" in accord with the Ada reference manual. A producer task calls the transmitter of its node and initiates the transmitter. The I/O processes of consumer tasks are accept actions in accord with the Ada reference manual. A consumer task calls the receiver of its node to deliver the specified message.

C.Berggren, US

Is this a concept only or have you actually implemented it? Does your development map onto military standards intended for task communications in future embedded distributed systems?

Author's Reply

The concept outlines the result of a study which analyzed the problems of task interactions in distributed architectures. The result of the study is a proposal which describes the structures, the functions and the functional layers of a powerful autonomous communication system usable in distributed architectures and the adaptation of this system to the program language Ada. The adaptation of general communication and/or protocol standards is part of a future project phase.

FUNCTIONAL PROGRAMMING LANGUAGES AND MODERN MULTIPROCESSOR ARCHITECTURES: THE EMSP EXAMPLE

J. DENNIS SEALS

GERALD W. GRUBE

Bell Laboratories
Whippany, New Jersey 07981

Summary: Future military systems will require a mix of signal, data, and control/decision processing at throughput levels ranging from 0.1 to 4 billion operations per second. Estimates of the embedded software to support these systems range from 0.5 to 10 million lines of code. Early in the 1980, the Navy recognized the severity of these problems and awarded AT&T Bell Laboratories a contract to design a revolutionary new computer architecture that would meet their signal processing needs through the year 2000. The result was the EMSP architecture that combines the programming simplicity of a functional language with the power of a modular, scalable, heterogeneous multiprocessor. The glue that binds these two attributes is a data-flow control mechanism that provides a transparent interface between the application user and the machine hardware. This paper will report on this successful integration of a functional graph language and a multiprocessor architecture.

Index Terms - Multiprocessing, Data-flow, Graphical Languages, Functional Languages

I. Introduction - The Problem

The requirement on future military systems to detect, engage, and defeat threats at ever increasing ranges and under hostile environments places unprecedented demands on the processing systems. The increased sensitivity and resolutions of new sensors, the need for low false alarm rates, and complex fusion/asset management algorithms demand a mix of signal, data, and decision processing at throughput rates that can easily exceed one billion operations per second. These requirements are forcing computer architects to come to terms with two extremely difficult problems - (1) designing a modular, scalable heterogeneous multiprocessor, and (2) developing a methodology for programming, testing and supporting embedded software.

The need for a multiprocessor architecture is readily derived. There is no existing or proposed uniprocessor that is capable of the required throughput. Even if such a feat were technically possible, it would have drawbacks from the standpoint of scalability and fault-tolerance. Since data, signal, and symbolic processing have their own special hardware requirements, it is clear that a heterogeneous architecture will be required. Unfortunately, complex multiprocessor architectures have been notoriously difficult to program, and that brings us to the second problem - software.

Estimates of the embedded software required to support programs such as ATA, ATF, and LHX range from 0.5 to 10 million lines of code. Current estimates for the cost of developing a line of high level code for embedded programs range between $200 and $2000. A 500,000 line program may require over 500 staff-years to develop and test. But this is only one part of the software problem. The cost of maintenance and support can represent as much as 80% of the life-cycle cost. Simple arithmetic shows that future software projects could cost billions of dollars unless new approaches are found to mitigate development, test, and support costs.

In this paper, we will present an architecture (language, compilers, run-time executives, and machine hardware) that meets these requirements and substantially reduces software cost. The machine described is not a laboratory curiosity, but a fully supported, mil-qualified production system.

II. System Concept - Designing the Next Generation Architecture

In early 1980, the U.S. Navy recognized that a revolutionary new approach to processing and software development was needed to meet future embedded processing requirements. They awarded AT&T Bell Laboratories a contract to create a new generation of computer architecture called the Enhanced Modular Signal Processor, EMSP [1]. The design goal for the project was straightforward - design a multiprocessor architecture that would meet the Navy's signal processing requirements through the year 2000, while significantly reducing software development and support costs. Implicit in this goal was the need for an open architecture that was scalable from a few processing elements to tens of processing elements to encompass the range of applications. The architecture had to retain software and system compatibility across all configuration, and provide a highly fault tolerant and supportable environment.

In order to meet this ambitious goal, the EMSP design team identified three major design concepts, all departures from traditional processing architectures. The first concept was the specification of a functional graph language as the means of capturing applications. This approach permits the application engineer to define the processing and flow of data as a data-flow program - a kind of formalized and highly structured flow chart. Data-flow programs are directed graphs, where the graph nodes represent processing functions, and the graph arcs represent the flow of data or synchronization tokens. Data-flow application graphs are not only simpler to develop than conventional procedural programs, but they also provide a natural paradigm for concurrent processing.

This departure from classical programming was not made lightly. Although functional programming languages have been around since the 1960s, there has been little success in achieving "real world" application. Much of the early work focused on proving program correctness. Subsequent efforts addressed the use of functional languages as a means for formal specification and validation of program requirements. Recent work has focused on the use of functional language as a means of application capture, but the results have been limited. One major reason for the limited success and slow acceptance is the incompatibility between functional languages, and architectures developed for procedural languages.

This led to the second major design concept, namely the design of a coarse-grain, dynamic, data-flow control mechanism [2]. This mechanism was implemented via a run-time executive called the EMSP shell. The shell dynamically manages queues and schedules nodes to processors when all node inputs are available. The shell implements graph arcs as dynamic first-in, first-out (FIFO) queues, and nodes as instruction streams - a set of processing instructions specifying inputs, outputs, and functions to be performed. When a queue reaches threshold (i.e., it has enough data to execute a node), a message is sent to the scheduling function. When all input queues for a given node reach or exceed threshold, the node's instruction stream is sent to any free processor for execution. This control mechanism naturally supports concurrent processing on a heterogeneous multiprocessor. This concepts frees the application user from machine dependent functions such as memory assignment and management, concurrent task scheduling, and task binding.

The third design concept was the specification of a open, modular architecture that can be configured for a wide range of applications. The machine architecture consists of four basic elements: a scheduler, a command processor, global memory, and node processing elements. The command processor supports graph management, system control, performance monitoring, and fault management. The scheduler supports the dynamic scheduling and assignment of nodes to processing elements. The global memories are intelligent repositories for for managing queues and node instruction streams. Processing elements interpert node instruction streams and execute primitive functions on the specified data. Processing elements come in many types, such as floating point signal processors, I/O processors, and data processors.

In the following sections we will describe the basic attributes of the graph language, and show how an application is captured as a graph. A brief overview of the functions of the run-time executive shell and machine architecture will be provided to show their interaction with the graph language.

III. The Graph Programming Language

The cornerstone of the EMSP architecture is its functional programming methodology. It is based on the Common Operational Software (COS) concept [3] developed at the Naval Research Laboratory. This methodology separates the application program into two components: the command program and the application graph. The application graph defines the underlying application processing, while the command program controls the interaction between graphs and external I/O.

The command program controls the starting and stopping of graphs, external I/O, graph parameters, interactions with the pilot and other tactical systems, as well as error responses and processor status. For example, the command program controls mode changes, revisit times, overload management, built-in-test, and diagnostics. It is important to note that since the control is not woven into the fabric of the processing flow, it is simple to tailor application processing dynamically.

The application graph is a radical departure from classical, procedural, programming languages such as Fortran, Ada, Jovial, and C. It is a functional language based on the directed data-flow graph. This concept is similar to the conventional flowchart used by programmers to structure their code development, but differs in two ways. First, the structure is more formal and rigorous. Secondly, the graph itself is the program.

There are three basic components to every application graph: nodes, queues, and auxiliary graph entities. The nodes defines the underlying processing function (i.e., FFT, data_merge, search_tree) that is to be performed on the input data. Queues manage the flow of information between nodes. Auxiliary graph entities are data structures that are primarily used to control, initialize, or direct the flow of information into and out of a node.

Nodes define the underlying application process. A node consists of three basic components: a primitive, a primitive interface procedure, and ports. The primitive defines the processing function to be performed on the input data. It is highly machine dependent code, and similar to the private part of an Ada package in that it cannot be modified by the application user. The primitive interface procedure defines the interfaces between the primitive and the nodes inputs. This code is similar to the public part of an Ada package. Ports are logical valves to which data queues are attached. They are controlled by parameters that determine when a queue is to be turned-on or turned-off.

The queue provides the primary data storage and transfer mechanism between two nodes. A queue can be conceptualized as an elastic FIFO buffer. A queue has one source node that produces (writes) data into the queue, and one sink node that consumes (reads) this data. This approach greatly simplifies graph capture, and ensures that the graph is free from side-effects. Multiple copies of a queue can be realized via a replicate node that produces multiple, independent copies. The language recognizes two basic type of queues: data queues and trigger queues. Data queues pass data that are needed for the underlying application process. Trigger queues pass synchronization information.

In addition to nodes and queues, one more component is needed is needed to complete an application graph - the auxiliary graph entity. These components are data structures (expressions, literals, or values) that specify parameters, such as the number of taps in a filter, the number of entries in a threat file, or the current operation mode. Auxilliary graph entities fall into two categories: graph variables, and graph instantiation parameters. Graph variables are expressions or literals that are declared within a graph or command program. They can be read or written by the originating graph or command program, but its current value can only be read by graphs to which they are passed. Graph instantiation parameters are constants that are evaluated at start time and passed by value to a graph.

In the next section, we will examine the tools and techniques for the schematic capture and compilation of application graphs.

IV. Programming with Graphs

The EMSP graph development tool set (see Figure 1) provides a rich and powerful environment for the capture, testing, documentation, and compilation of application graphs. It supports a highly structured methodology for software development that naturally partitions an application into smaller units with well defined interfaces. This attribute permits the application programming to be allocated to groups with specialized talents and well defined responsibilities.

The first step in the capture of an application graph is its schematic capture. The Graphical Editor (GRED) is a language-directed, interactive editor that enables the user to create, modify, and annotate graph objects (subgraphs, nodes, queues, etc.). It supports a top-down, hierarchical development in which the development of a high-level graph is the first step. This graph consist mainly of subgraphs, the flow of information among these subgraphs, and the system interfaces (see Figure 3). The subgraphs are simply black boxes at this stage. This top level view of the application is usually defined by system engineers, who address high level interactions and system interfaces. Once this graph is completed, the subgraphs are assigned and defined by sensor and subsystem processing specialists. Definition of each sensor/subsystem subgraph can be carried out concurrently and independently. There is no need for the application engineers to become expert in high level languages such as Ada, or to convey their application to a programmer who is.

Graphs are created on a high-resolution, bit-mapped terminal. The principle mode of input is via a three button mouse that permits the user to select graph objects, attributes, viewing levels, and edit functions from pop-up menus. The keyboard is required only for naming objects, graph files, or filling in templates. The graph capture process is relatively fast, allowing graphs of several thousand nodes to be captured in a few weeks or less. The editor also provides some run-time syntax checking, catching many errors during the edit process.

The output of the graph editor is passed to the graph source generator, which creates an intermediate source language that is more amenable to automatic document generation and syntactic debugging.

The graph parser accepts the intermediate source code and performs extensive syntactic and semantic error checking. It ensures that all data typing, and queue initialization is defined and compatible. The output of the graph parser can be passed to any of a number of compilers.

The graph tool set currently supports compilers for a system level simulator, a graph simulator, and the target machine hardware. The system level simulator allows the user execute a graph, without explicitly evaluating the nodes. Node execution is treated as a black box with a specified execution time. The graph simulator is used to debug the application function. It executes the underlying function of each node, though not necessary in real-time nor concurrently.

Studies have been performed to evaluate the relative improvement provided by functional graph programming over conventional languages. These studies was performed using real-world applications, complete with I/O and control. The result. show that on average, the ECOS graph language provided a five-fold decrease in the code and test phases over traditional high-level languages. Although there are no studies to evaluate improvements in the support and maintenance phase, the ease with which a person can grasp the underlying process of a complex graph does suggest that a significant savings may occur.

V. The Run-Time Executive - Making a Machine Transparent

The EMSP data-flow executive software provides a transparent interface between the application program and the machine hardware. This executive is based on a coarse-grain, data flow paradigm that schedules nodes based on the availability of their data inputs. The executive provides three basic functions: queue management, node scheduling and assignment, and node execution. Figure 3 show the relationship and information passed between these functions.

The queue management function creates, monitors, and deletes queues. Queues are maintained as dynamic FIFO structures, with associated attributes such threshold and capacity. The threshold is a user defined value or expression that specifies the minimum number of items needed in the queue for its sink node to execute. When the number of elements in the queue meets or exceeds threshold, a threshold message is sent to the scheduling function. No further threshold messages are permitted from that queue until its node completes execution. When one or more queues are input to a node, each queue must exceed threshold before a node can be executed. The queue capacity is primarily a warning parameter that the queue will soon reach an over-capacity status and overload management should be invoked.

The scheduling and assignment function determines when all of the inputs to a node are available, and assigns the node to an available processor. It performs this function through the use of two tables. The queue-to-node table is basically a connectivity map that specify the queue's produce and consume node ports. Each queue connects at most two nodes - one that produces data into the queue, and one that consumes data from the queue. The queue-to-node table provides a pointer to a node status table that maintains a list of all inputs for each node and their threshold status. When all the necessary inputs are available for a given node, it is assigned firing status. It is priority queued, being assigned to the first available processor that has the resources to execute the node. This assignment is completed by sending the node's instruction stream to the node execution function.

The node execution function executes node instruction streams. This instruction stream specifies where to obtain the input data, rules for computing memory storage, and the functions to be performed on the inputs. It also specifies where to output the data and what input data is to be consumed (destroyed). When this is completed for a given instruction stream, a completion message is sent to the scheduling function. This message informs the scheduler that the processor has completed the node execution, and is free to execute another task. It also permits that node to be scheduled again, once its inputs are available.

In a typical application this process is carried on concurrently for thousands of nodes and queues. The use of dynamic assignment not only simplifies load balancing and parallel processing, but it also provides an effective means of fault recovery. If a processor fails during execution, its nodes are simply reassigned to another processor, as the inputs are not consumed until the node has successfully executed.

V. THE EMSP Machine - Scaling from MOPS to GOPS

The EMSP architecture is an open, modular architecture that is capable of supporting a wide variety of configurations and platforms. This versatility is achieved, in part, by a set of system building blocks called functional elements. Each element uses the same protocols and internal data/control interfaces. The basic EMSP architecture consists of four types of functional elements: a scheduler, a command processor, global memories, and processing elements (See Figure 4). These functional elements are combined into a system by separate data and control transfer networks.

The command processor supports graph management, system control, performance monitoring, and fault management. It executes the command program part of the application program. The scheduler maintains the execution status for all the nodes in a graph, and assigns executable nodes to available processing elements. It maintains the queue-to-node table, the node status table, and the available processor queue. Both the scheduler and command processor may be configured as a doubly redundant function element for improved fault tolerance.

The global memories are intelligent repositories for the storage and management of queues, node instruction streams, and auxiliary graph entities. They dynamically allocate and de-allocate memory for queue. They send queue threshold and capacity messages to the scheduler, and instruction streams and data to the processing elements. Each global memory consists of a control processor module and one or more bulk memory module.

The processing elements decode node instruction streams and perform the primitive function(s) specified. The EMSP architecture supports a variety of processing elements such as a floating-point signal processor, I/O processors, special signal-conditioning preprocessors, and data processors. Processing elements typically consist of a control processor that decodes the instruction stream and manages internal memory. Separate, special purpose processors execute the primitive functions as directed by the control processor. This approach permits simultaneous setup, execution, and breakdown to occur within a processing element.

The functional elements are interconnected by separate data, control, and built-in-test networks. The data transfer network consists of a non-blocking, asynchronous, buffered switch network that permits simultaneous transfers between functional elements. The control messages and BIT data are transferred over separate busses.

VII. Conclusions - The Proof is in the Deployment

Over 500 staff-years of development have made the EMSP architecture into a successful and mature product. It is the Navy's standard signal processor, the AN/UYS-2. It is a fully mil-qualified, production system with extensive documentation and support software. Its ECOS language has been demonstrated to provide a five-fold reduction in software costs over traditional high-order language systems. It is currently being used to capture complex real-world applications consisting of thousands of queues and nodes. The architecture provides between 40% and 90% execution efficiency across all processing elements for a broad range of applications. It is targeted for numerous military programs. A second generation architecture is scheduled for production in two years, and will provide a five-fold improvement in throughput density, with enhanced data processing capabilities. A third generation architecture (MGP) is currently in the research and development phase, and promises another factor of five increase in throughput density. It will also provide additional data and decision processing capabilities. The MGP architecture will provide over four billion, 32-bit floating point operations per seconds (signal processing) and 100 million instructions per second (data processing) in a 1.5 cubic foot enclosure.

In conclusion, this architecture has achieved all its initial goals through the marriage of a functional programming language, a data-flow executive, and a modular, heterogeneous multiprocessor.

REFERENCES

[1] EMSP/ASP Common Operational Support Software Methodology, Analytic Displines Inc., Version 3, May 1984.

[2] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," Proceeding of the IEEE, Vol. 75, No. 9, Sept. 1987, pp. 1235-1245.

[3] LCDR W. L. Hatcher, "EMSP: The Navy's Next Generation, Real-Time Signal Processor," Southcom, January 1984.

[4] J. D. Seals and R. R. Shively, "VLSI Signal Processing," First Edition, New York, IEEE Press, 1984, pp.421-425.

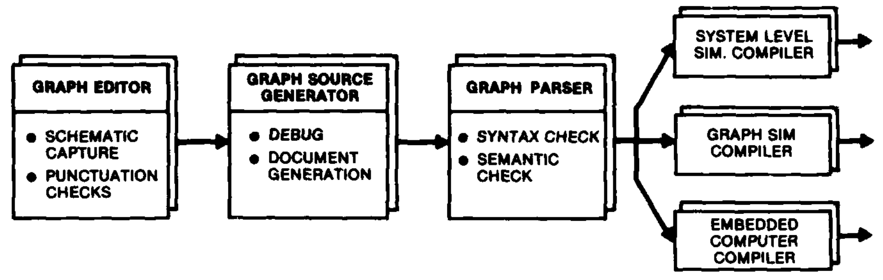


FIGURE 1 — THE BASIC GRAPH TOOL SET

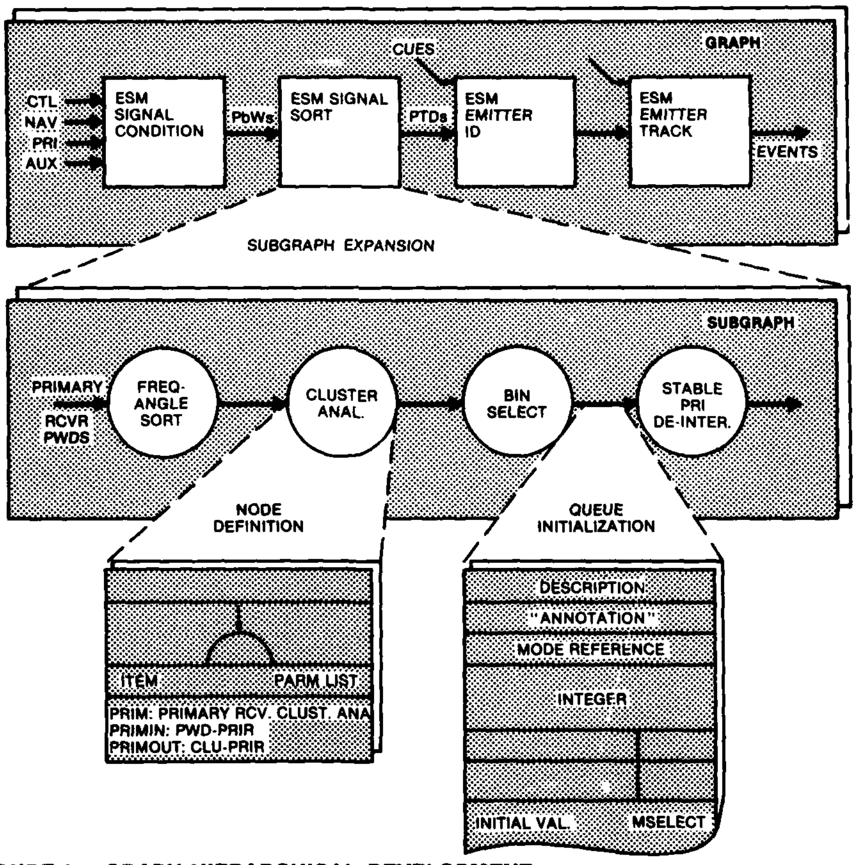


FIGURE 2 — GRAPH HIERARCHICAL DEVELOPMENT

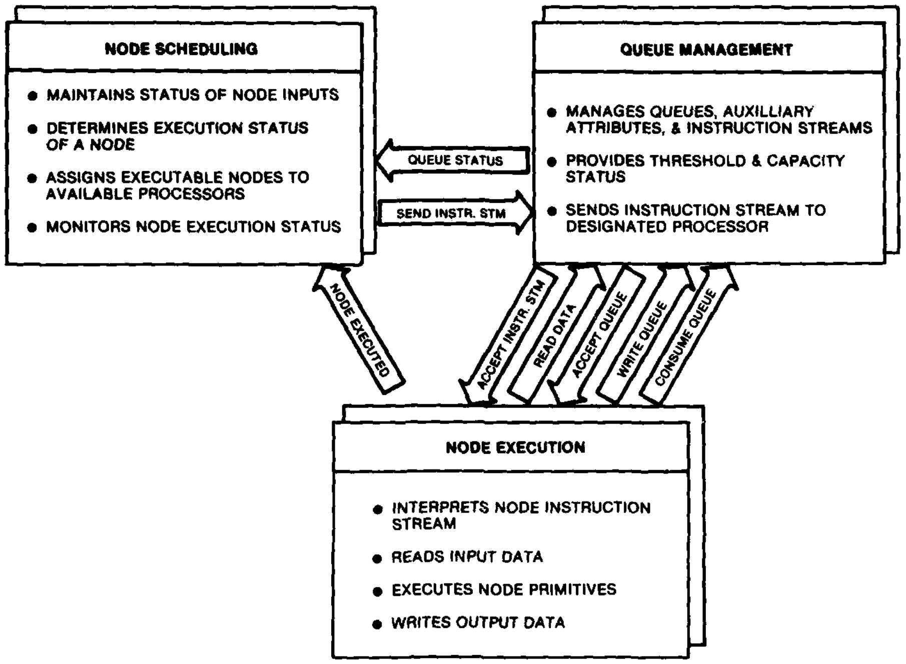

NODE SCHEDULING
● MAINTAINS STATUS OF NODE INPUTS
● DETERMINES EXECUTION STATUS OF A NODE
● ASSIGNS EXECUTABLE NODES TO AVAILABLE PROCESSORS
● MONITORS NODE EXECUTION STATUS
QUEUE MANAGEMENT
● MANAGES QUEUES, AUXILLIARY ATTRIBUTES, & INSTRUCTION STREAMS
● PROVIDES THRESHOLD & CAPACITY STATUS
● SENDS INSTRUCTION STREAM TO DESIGNATED PROCESSOR
QUEUE STATUS
SEND INSTR. STM
NODE EXECUTED
ACCEPT INSTR. STM
READ DATA
ACCEPT QUEUE
WRITE QUEUE
CONSUME QUEUE
NODE EXECUTION
● INTERPRETS NODE INSTRUCTION STREAM
● READS INPUT DATA
● EXECUTES NODE PRIMITIVES
● WRITES OUTPUT DATA

FIGURE 3 — RUN-TIME EXECUTIVE FUNCTIONS

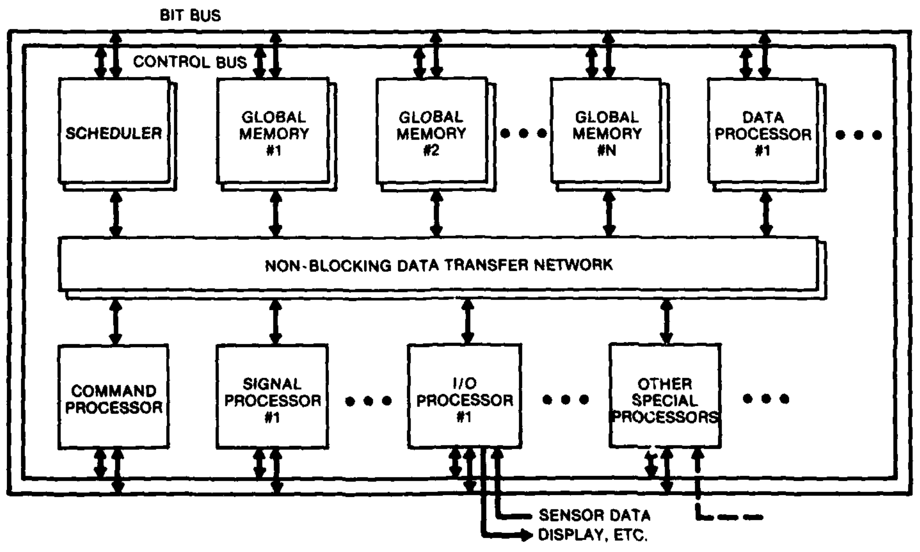


FIGURE 4 — MACHINE ARCHITECTURE

DISCUSSION

G.Paris, Fr

Aren't the two last papers a solution to the question "Can Ada fly missiles"? We must have standardized Ada, a methodology and communication tools if we hope to realize software reusability. What is your opinion?

Author's Reply

I agree! The purpose of the graph language and tools presented here is to supplement rather than replace Ada. The graph language is a very effective program design language (PDL). It also simplifies coding and testing by automatically generating the interface or "public" portion of an ADA package.

M.Stoll, Fr

When you use this tool to update a system, how do you generate the modification and how do you manage the configuration?

Author's Reply

Updates and configuration management for graphs are performed in much the same manner as conventional programs. One can add, delete and/or modify graphs, subgraphs or nodes. In many ways the update process is simpler because the application process is easier to grasp in a picture (graph) form vs. actual code. Also, graphs have good *locality of effect*. That is, changes tend to impact only very local regions in the graph. Graphs and subgraphs are stored as files that can be linked and edited in a manner similar to other software files.

P.Tillman, UK

Will graphical languages such as yours make Ada obsolete in the medium term?

Author's Reply

The graph language presented in this talk supplements, rather than replaces, languages such as Ada. It is important to remember that the primitive kernels of nodes, the I/O procedures and the command program(s) are programmed in a HOL (Ada or SPGN). In practice, approximately 50+% of the traditional HOL code is captured using the graph language. In many ways, this corresponds to the relationship between HOLs and assembly languages.

W.Mansell, Ge

The effort saved using the GPL method compared to HOL seems astonishing. Can you comment on how this reduction is achieved?

Author's Reply

There is no single contribution that accounts for this dramatic improvement; but rather it is the result of many attributes. These include:

(1) Pictures (graphs) are easier to produce and comprehend
(2) Data flow graphs are side-effect free.
(3) Data flow graphs naturally capture parallelism (@ coarse-grain level).
(4) Graph provides good locality of effect.
(5) They permit higher levels of abstraction.
(6) The graph compiler automatically generates *correct* interface code (the public portion of an Ada package/ procedure).

It should be noted that application primitives, command programs and I/O packages must still be written in Ada. Thus, the five-fold improvement (of the graph language) over HOLs translates into about a 50 to 60% improvement in overall lifecycle costs.

Real Time Expert System for Onboard Radar Identification.

Thierry SCHANG
AI Electronic Warfare System Engineer
THOMSON-CSF
Radar, Counter-measure, Missile division
178, bvd Gabriel Péri
92242 MALAKOFF CEDEX - FRANCE

François FAGES
Expert System Laboratory Manager
THOMSON-CSF
Laboratoire Central de Recherche
Domaine de Corbeville
91401 ORSAY

Summary :

The use of Expert System techniques for radar identification in counter-measure systems has been widely discussed in the literature. The main advantages of such a system are that it is easily adaptable to the ever changing operational field and that it can identify modern radars with their increasingly complex signals. These systems are however very demanding on computing ressources and their adaptation to on-board, real-time processing is not straightforward. In this paper, the application under consideration is first briefly described. The constraints that arise out of its real time operation are then detailed along with how each one can be eliminated. The application of these concepts to a real time radar identification function is then described. Finally, future objectives in the integration of Artificial Intelligence techniques for on-board processing are discussed.

1. Introduction

The use of Expert System techniques for radar identification in counter-measure systems has been widely discussed in the literature. The main advantages of such a system are that it is easily adaptable to the ever changing operational field and that it can identify modern radars with their increasingly complex signals.

One possible use for a radar identifier is onboard an aircraft to instantly give the pilot valuable information about the immediate threat situation. However, Expert Systems are usually very demanding on computer ressources and, the change from a large computer laboratory environment to a small size computer in a real time environment is accompanied by obvious problems. Both the application structure and the computer tools must be reviewed in order to satisfy the real time operational constraints.

In this paper, the application under consideration is first briefly described. The limitations that arise out of its real time operation are then detailed along with how each of these limitations can be overcome. The application of these concepts to a real time radar identification function is then described. Finally, future objectives in the integration of Artificial Intelligence techniques for on-board processing are stated.

2. Countermeasures Environment

In modern defense techniques, an aircraft mission must be backed up by appropriate electronic procedures. Success very much depends on knowledge of the enemy's electronic order of battle (EOB). For these procedures to be successful, the first step to be accomplished is to identify the th.eats and then to decide for an appropriate action to be undertaken.

The threats are identified through their associated radars by processing the electro-magnetic information gathered by an airborne Electronic Warfare sensor. The sensor measures the characteristics of the emitted

This research is supported in part by the Direction des Recherches, Etudes et Techniques, (DRET), FRANCE.

pulses and carries out preliminary sorting to bring together pulses corresponding to the same radar signal. The identification is then performed by comparing the incoming signal with a predefined library of signatures. Figure 1 gives the overall structure of an Electronic Warfare System.

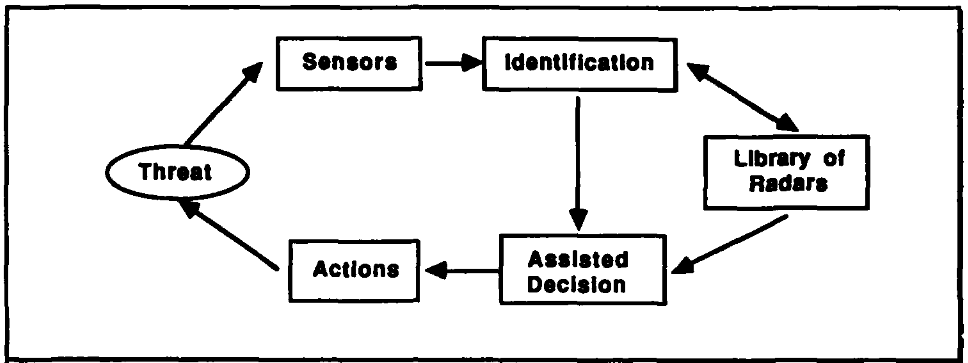


Figure 1 : Structure of an Electronic Warfare System

3. The Identification Function

3.1. Identification Problems

However, identification gets harder as the complexity of modern emissions increases. Moreover, in case of conflict, the threat characteristics are very likely to change as new or undiscovered radar operating modes are used, requiring a system which is easily adaptable. Conventional identification algorithms are often unsatisfactory and difficult to adapt to a changing operational field. As human experts exist who are able to carry out identification, expert system approaches to the problem have been taken and have been discussed by various authors in [1,2,3].

The first phase of our project was to design a prototype to demonstrate the validity of the approach using powerful Artificial Intelligence techniques without any limitations on computing power, response time or memory space requirement. The resulting prototype is briefly described in the next section, more details are available in [1].

As the first phase was completed successfully, the next step was to take into account the real time limitations to evaluate the possibility of integrating the system directly into a fighter with very limited computing power.

3.2. Non Real-time Identification

The signals to be identified are gathered during flight by an ESM sensor. However, identification takes place only when the flight is finished, in a conventional computer center, with a large amount of computing ressources. The principles of the identification process were directly derived from interviews and observation of a knowlegeable radar identification technician (i.e. the domain expert). The main development goal was to reproduce as closely as possible his reasoning process.

3.2.1. Principles

Two aspects of the human expert are taken into consideration. First, WHAT are the sources of knowledge that are used, and second, HOW is all this knowledge used to perform the identification ?.

The sources of knowledge used by the expert are of three kinds. First, the knowledge of existing radars and there characteristics,which is stored in a conventional data base management system. Second, the existence of special algorithms used by the expert to find out special features of the signal. Last, the expert's specific knowledge relative to the sensor limitations, to the radar operational functions and to the geographical or political operational environment.

The identification process is divided into four main steps. The first is a qualitative analysis of the incoming signal. It ensures detection of defective sensor working and possible corrective action using special algorithms. The second is data base selection in which a few likely candidates are extracted from the library for identification. In the third step, each candidate is thoroughly analyzed to find the one that matches the signal best. A fourth step may be activated if the above fail to give a satisactory answer. It consists in hypothesizing about operating conditions of the sensor, radar or about any known phenomenon in the environment. Figure 2 gives the overall structure of the non real-time expert identifier.

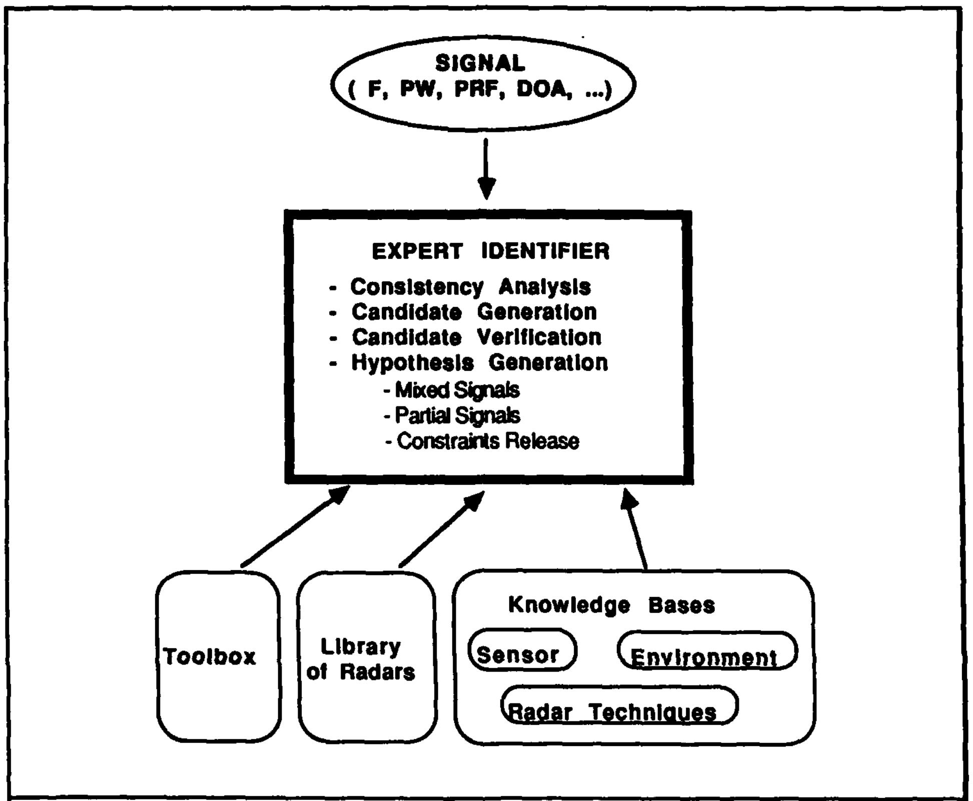


Figure 2 : Expert Identifier Structure

3.2.2. Trials

The use of real data ensures that the trials are not biased by an incomplete simulation of all the operational problems. Various prototypes have been implemented on various tools. The best results were obtained using the expert system compiler SPOCK (described later and in [4,6]). An identification can be performed in an average time of less than one second on a microvax II. The possibility of integrating the system in a real time environment can therefore be considered.

4. Real Time Requirements

4.1. Constraints

All the requirements inherent in an aircraft operation must be taken into consideration. They can be placed in three different categories which are :

- Computing resources constraints
- Software Integration constraints
- Application Design constraints

Each of these is discussed in the following sections.

4.1.1. Computing Resources Constraints

Computer Resources on a fighter are limited by the size and weight available (a few printed circuit boards only), and by the environmental constraints. In most cases no disk storage is available and therefore no virtual memory system can be implemented. Memory and CPU usage must therefore be limited drastically.

Furthermore, an airborne system must be in operation continuously during the mission, and procedures such as memory garbage collection (used in most of the Artificial Intelligence systems today to recover the memory no longer used by the program) are therefore not acceptable.

4.1.2. Software Integration Constraints

If the expert system must be added to existing equipment, additional constraints appear. It must then be integrated to an existing real time operating system and to conventional software and hardware for data communication between the different processes. The use of a common programming language is therefore recommended for all parts of the system (C, ADA, LTR3, ...)

4.1.3. Application Function Constraints

Some operational constraints of the identification function were not taken into account. In a fighter environment, the system must give a usable result in a given time and must be able to process constantly evolving data.

4.2. Solutions

To overcome all these limitations, various aspects requiring major changes are involved. Waiting for more powerful computers to appear is not the complete solution to the problem. A high performance tool must be used and the application must be deeply restructured.

4.2.1 A High Performance Expert System Shell

Part of the solution is to use a specialized expert system generator to achieve a trade-off between high-level functions and speed. An example of such a system is SPOCK. A detailed description of SPOCK is given in [4,6,9]. Its main functions and contributions to real-time operation are the followings:

SPOCK is an expert system generator that produces a conventional program in a given language (Lisp, C or ADA). The program can then be compiled for fast execution on the target machine. No extra time is thus spent by the language interpreter as in a lot of expert system shells. This feature also facilitates the integration of the expert system in a real time environment and to conventional software and hardware.

The heart of SPOCK also manages its own memory resources, it does not therefore need to be stopped for memory recovery (garbage collection) and so is available for continuous operation.

It includes some functions relating to the faculty of reasoning in the time domain, enabling it to consider the constant updating of the information. Mechanism is also provided for asynchronous input of data.

SPOCK is being developped by the Research Central Laboratory of THOMSON-CSF in three versions that corespond to different function level in order to integrate the major artificial intelligence paradigms such as forward/backward chaining, first order predicate calculus, objects manipulation, non-monotonic reasoning and multiple hypothesis manipulation.

4.2.2 Need for Restructuring the Application

The application itself must be redesigned to take into account some of the real-time limitations.

To be able to provide a usable result in a given time, the system should have different lines of reasoning enabling it to process the same data with various computing costs and qualities so that the system can choose the most appropriate one depending on the computing time available. This idea was first mentioned in [5], it can be achieved by structuring the knowledge base at levels which are different and that are activated according to available time. Examples are given in the next section.

The knowledge base must also be able to process constantly evolving data. That is, it must be able to cancel part of the reasoning when some of the data changes and to take into account the operational meaning of successive signal states.

5. Real-time Radar Identification

To prototype the real time system, a simulation of the real time data flow is done on the basis of real data gathered during flights with ESM sensors. In the next section, a description of the simulation is given. The problem of hierarchizing the knowledge base is then discussed.

5.1. Simulation Structure

The overall structure of the simulation is given in figure 3.

The signal constitution element carries out preprocessing of the measured data. This consists in creating and updating signals in the common workspace. Techniques used are derived from various clustering techniques.

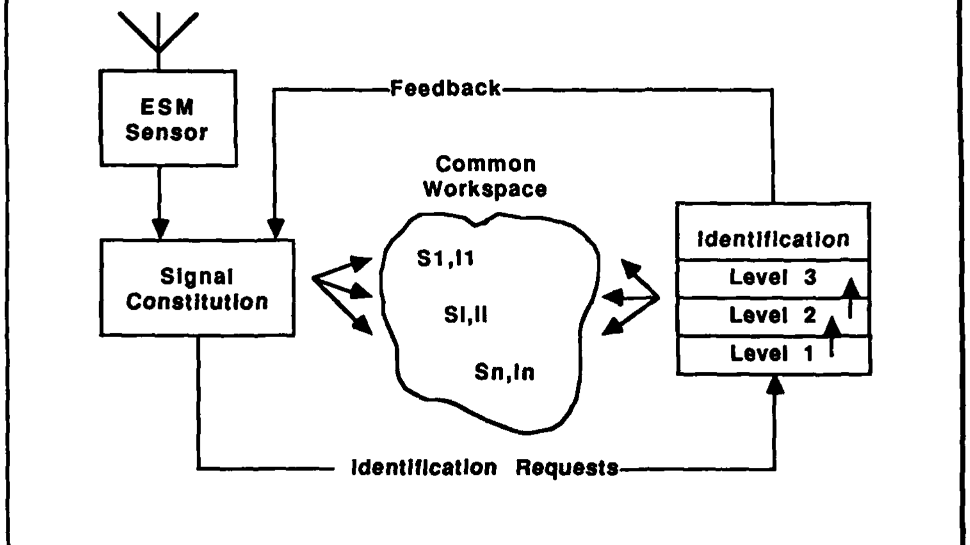


Figure 3 : Simulation Structure

When a signal is considered sufficently good, an identification request is sent to the identifier on an asynchronous basis. Furthermore, when a signal undergoes significant modifications, a re-identification request is sent to the identifier.

5.2. Multiple Levels

The knowledge base of the identification process is organized in several levels. This enables tasks to be performed according to priority. That is, the expert system first processes all level one requests. When no more inferences are possible at this level, it switches to the next level, until a lower level task is initiated. Switching from one level to the other, can be done asynchronously between the firing of any two rules.

A description of typical processing at each level follows:

Level 1 : Low level or high priority processing
- systematic pre-identification of all identification requests from the signal constitution element
- high priority identifications determined in the pre-identification phase.

Level 2 : Medium complexity processing
- Analysis of complex quality criterion that requires more processing power.
- Confirmation of pre-identifications.

Level 3 : High complexity processing
- **Execution of complex cleaning algorithms when some criteria value requires it.**
- **generation of complex hypothesis**

Different lines of reasoning for the same problem may be implemented at different levels. In this case the identification achieved at a given level can be refined by the results of a higher level.

Initiation of a task at a given level is either by a request from the signal constitution module or by any rule in the knowledge base. The processing of a signal can therefore dynamically move from one level to another.

5.3. Evolving Data

Constantly evoluting data can introduce problems in the identification process. A modification of the signal may contradict previous inferences. The system must therefore be able to cancel these inferences and start a new identification. An initiated task at a high level not yet executed may also be cancelled when additional information arrives.

The use of non-monotonic logic can ease the implementation of such a feature. For example, a rule might say:

```
IF
  a quality criterion number 1 is lower than a given threshold
THEN
        several signals are mixed
  AND   initiate special processing at level 3
```

When this rule fires, the special processing is not executed immediately; subsequent information can change the value of the criterion and therefore the special processing is deactivated automatically.

5.4. Trials

The prototype of the system is written on a microvax II computer using SPOCK as an expert system generator in LeLisp language, conventional algorithms are coded in PASCAL.

Particuliar attention will be paid to computer resources used by the system. As trials are carried out with real data representative of a very dense radar environment, the results will give a good insight into the required cpu performance and storage. It will then be possible to precisely dimension the computer that should be put into the aircraft for a given identification performance objective.

6. Conclusion

The main constraints associated to real time integration of expert system techniques has been stated. To use more powerful computers is not enough, performance can and must also be gained on the software side of the problem first by using a high performance expert system generator as SPOCK and second by thoroughly restructuring the application itself to adapt it to the required real time environment. An example has been given of an expert system for real-time radar identification derived from a non real-time expert system. The combination of these techniques will shortly make it possible to integrate the expert system on an operational equipment without having to rewrite the entire application using conventional programming techniques. The high degree of flexibility of these techniques will therefore enable more powerful systems to be installed onboard aircrafts.

The long term objective is to further facilitate the use of these techniques by integrating directly to an ADA programming environment, an object manipulation layer and a production rule layer.

7. References

[1] T. SCHANG, "Threat Identification : An Artificial Intelligence Approach.", 52nd Symposium AVP AGARD, Sept 86, Florence, Italy.

[2] H.J. WELLS,"The Application of Intelligent Knowledge Bases System 6. H.J.WELLS, "The Application of Intelligent Knowledge Bases System Techniques to Emitter Recognition", 52nd Symposium AVP AGARD, Sept 86, Florence, Italy.

[3] M. GIUBBOLINI, F. REVERBERI, "ESM Identification Based on AI Techniques", 52nd Symposium AVP AGARD, Sept 86, Florence, Italy.

[4] J. LACROIX, G. BERTHON, F. FAGES, "SPOCK, un outil pour l'intelligence artificielle temps réel", 55th Symposium AVP AGARD, April 88, Cesme, Turkey.

[5] L. WRIGHT, M.W. GREEN, G. FIEGL, and P.F. CROSS, "An Expert System for Real-Time Control", IEEE Software, March 1986.

[6] F. FAGES : "On the Proceduralization of Rules in Expert Systems", First France-Japan Symposium on Artificial Intelligence and Computer Science, Tokyo, Octobre 1986, dans "Programming of Future Generation Computers", North-Holland, Eds M. Nivat et K. Fuchi.

[7] TH GREER "Artificial Intelligence : A New Dimension in EW", Defense Electronics, Octobre 1985, P108-128.

[8] K. FRANKOVITCH, K. Pedersen, S. Bernsteen, "Expert System Applications to the Cockpit of the 90's", IEEE AES Magazine, January 1986.

[9] F. FAGES, "Rule Based Extension of Programing Languages : A Proposal to Integrate Expert System into Applications at the Target Language Level", submitted to the 8th International Workshop on Expert Systems and their Applications, May 1988, Avignon, France.

DISCUSSION

R.Marmelstein, US

My concern is that the SPOCK system may have trouble managing memory resources on an Ada based expert system. In many compilers, unchecked deallocation is implemented with varying degrees of effectiveness. This may result in ultimately running out of memory.

Author's Reply

Memory management in SPOCK does not rely on the Ada deallocation system. The inference engine uses Ada data structures of fixed size and does not need to be extended unless the memory is full.

SPOCK, UN OUTIL POUR L'INTELLIGENCE ARTIFICIELLE TEMPS REEL

J.P LACROIX ; G.A BERTHON ; F.FAGES ; P.REPUSSEAU
Thomson-Csf AVG
31 Rue Camille Desmoulins
92130 Issy-les-Moulineaux France

Thomson-Csf LCR
Domaine de Corbeville
BP 10
91401 Orsay France (F.Fages)

RESUME : Les systèmes experts embarqués doivent prendre en compte des contraintes temps réel sévères, qui, associées aux contraintes d'environnement de la machine d'exécution, obligent à rechercher des solutions nouvelles. La procéduralisation obtenue par compilation de l'algorithme de RETE vers des langages cibles symboliques, procéduraux ou orientés temps réel apporte une solution aux principaux problèmes des applications embarquées : Besoins de vitesse, contrôle de la mémoire, mariage des parties symboliques et des parties classiques des applications. Cette approche est soutenue par la famille des outils SPOCK (-LISP, -C, -LTR3, -ADA) *developpée par le Laboratoire Central de Recherche de Thomson-Csf*, qui offre des améliorations significatives par rapport aux produits actuellement disponibles. Une des applications de SPOCK dans Thomson-Csf est un système d'aide aux missions d'hélicoptères prenant en compte les menaces et le relief grâce au couplage à une base de données cartographique embarquable.

ABSTRACT : The avionic real time expert system designer will now receive help from a set of software tools developed in the Central Research Laboratory of Thomson-Csf. Through RETE algorithm compilation targeting conventionnal (C) or real-time (ADA) languages, the SPOCK tools family offers an improvement close to one order of magnitude in performance, rated to currently available tools. SPOCK is used in severall military divisions of Thomson-Csf ; this paper describes SPOCK and its application to an AI based mission help for helicopters.

1 CONTRAINTES DES SYSTEMES EXPERTS TEMPS REEL EMBARQUES

L'exploitation opérationnelle d'un système expert dans les applications militaires embarquées doit prendre en compte les spécificités de ces applications, et les machines d'exécution s'adapter aux conditions d'environnement sévères rencontrées.

Pour la partie système expert, on rencontrera les besoins suivants :

- Analyse de situation et prise de décision en temps réel au sens de l'avionnique (<0.1 S),
- Prise en compte au niveau raisonnement des aspects temporels et spatiaux,
- Capacité à traiter des informations tronquées ou incomplètes,
- Cohabitation/fusion avec des applications ou des traitements conventionnels (algorithmie/calculs/gestion d'entrées-sorties...),
- ...

Pour la machine embarquable, les contraintes sont d'offrir sous le plus faible volume possible, avec la consommation la plus réduite et dans des ambiances de températures et de vibrations exigeantes :

- Des ressources mémoires étendues,
- Des capacités de traitement très élevées en comparaison des besoins des traitements classiques.

2 SOLUTIONS ENVISAGEABLES

Pour répondre aux besoins de traitement on peut envisager trois approches :

2.1 FORCE BRUTE

La machine d'exécution présente la plus grande puissance de calcul possible (quelques MIPS, voire plus) en vertu de l'équivalence entre une inférence et quelques dizaines d'instructions classiques. Le système expert lui-même est écrit dans un langage conventionnel, Pascal ou C, éventuellement un langage temps réel, LTR3 ou ADA.

AVANTAGES :

- Pas de mécanisme coûteux de récupération mémoire à réaliser,
- Intégration aisée avec le reste de l'application, lui-même écrit dans un langage de ce type.

INCONVENIENTS :

- Environnement de développement symbolique très pauvre pour les langages cités,
- Difficultés ou lourdeurs pour réaliser certaines parties du système expert,
- Doutes sur les capacités de cette approche à réaliser des applications "symboliques" très performantes.

Exemple de réalisation : Système expert HEXSCON du SRI.

2.2 SYMBOLIQUE PURE

2.2.1 Sur machine classique

C'est une approche basée sur la philosophie machine précédente, avec l'emploi d'un langage de type symbolique (COMMON-LISP, PROLOG, ...) compilé pour de meilleures performances.

AVANTAGES :

- L'environnement de développement est très riche.
- Les performances peuvent être très bonnes sous certaines conditions (pas de récupération mémoire activée).

INCONVENIENTS :

- Ces machines n'ayant pas de mécanismes matériels de récupération d'espace mémoire doivent stopper tout traitement pendant des intervalles variables (dépendant de la taille mémoire installée et du problème traité) et s'avèrent peu utilisables pour des applications temps réel.
- Les interfaces logicielles avec les langages classiques sont mal commodes.

EXEMPLES : Stations de développement SUN (type3-XXX), TEKTRONIX,HP...

2.2.2 Machine à architecture dédiée

Ces machines sont construites sur des architectures spécialisées (association d'étiquettes aux données, récupération mémoire câblée, mécanismes adaptés aux particularités du langage, ...) pour les langages symboliques.

A condition de les débarrasser de leur environnement de développement pour en faire des machines d'exécution, elles peuvent prétendre au créneau IA embarquée.

AVANTAGES :

- L'identité machine de développement/machine d'exécution assure la répétabilité des performances du laboratoire vers l'application.

INCONVENIENTS :

- Les mêmes qu'au paragraphe. pour le logiciel
- La nécessité de réduire l'encombrement de la machine par des actions d'intégration poussée.

EXEMPLES :

- Compact Lisp Machine de TEXAS avec VLSI LISP (MEGACHIP).
- Machines SYMBOLICS avec le nouveau VLSI ("IVORY").

Pour ces produits, l'accès sans restrictions à des versions militaires n'est garantie par aucun des deux fabricants.

2.3 PROCEDURALISATION

Cette approche est développée par le Laboratoire Central de Recherche de THOMSON-CSF et appliquée dans les Divisions "militaires" du groupe. Son intérêt est d'éliminer deux points gênants pour les applications IA embarquées :

- l'interfaçage entre langages symboliques et classiques,
- la nécessité du récupérateur mémoire dans la machine.

La méthodologie s'appuie sur un outil de haut niveau, appartenant à la famille générique SPOCK, permettant de procéduraliser les systèmes experts.

AVANTAGES :

- gestion totale de la mémoire,
- langage cible C ou ADA : interfaçage facile avec la partie conventionnelle de l'application,
- portage performant sur les machines classiques.

RESERVES :

- l'outil est actuellement à l'état de prototype.

3 L'OUTIL SPOCK

SPOCK est issu de recherches au laboratoire "Systèmes Experts" du Laboratoire Central de Recherches de *THOMSON-CSF, menées sur la compilation des bases de connaissances pour répondre aux besoin des branches* militaires de THOMSON-CSF en systèmes experts temps réel.

La compilation des règles est apparue comme étant la clef de l'obtention de performances élevées. *L'approche suivie permet de procéduraliser totalement le langage de règles (logique du premier ordre)* dans le langage cible.

L'algorithme de RETE (principe de base de OPS-V, ART et Knowledge-Craft) a été étendu pour compiler totalement le parcours du réseau de filtrage dans le code correspondant au langage cible. Différents langages cibles peuvent être adressés ; LISP et C sont disponibles, LTR3 puis ADA sont en cours d'étude.

SPOCK est :

Un exécutif de systèmes experts répondant aux besoins des applications temps réel, grâce à deux services offerts:

- maîtrise de l'allocation mémoire sans nécessité de "garbage collector",
- primitives de communication avec l'extérieur.

SPOCK est héritier de :

- OPS-V
  - * prépondérance du chaînage avant dans les règles
  - * représentation des objets par des vecteurs
  - * filtrage par l'algorithme de RETE
- ART
  - * philosophie d'intégration de plusieurs paradigmes sur un noyau de type OPS-V
- ML
  - * langage de prototypage utilisé pour l'écriture du compilateur de règles

SPOCK a des points originaux :

- les règles portent sur des objets abstraits qui peuvent être représentés par n'importe quelle structure du langage cible (defstruct ou flavors en LISP, record en ADA, struct en C, ...).
- compilation totale des règles : le réseau de RETE n'existe pas à l'exécution, le parcours du réseau a été compilé,
- exécution "garbage-free" garantie, tant que les règles ne comportent pas d'appel explicite à LISP.
- règles munies d'une priorité numérique, soit statique soit dynamique, cad calculée à partir des variables du membre gauche de la règle.
- représentation souple des objets, adaptée à l'implantation ultérieure des mécanismes de gestion d'hypothèses.
- compilation optimale du quantifieur existentiel ; possibilité de réordonner les prémisses dans les règles pour optimiser les tests effectués à l'exécution, à partir de statistiques d'utilisation.
- appel d'une fonction de communication après chaque cycle d'inférence pour prise en compte d'événements extérieurs ou gérer un échéancier.

SPOCK est performant :

Une comparaison de SPOCK avec d'autres outils, pour la résolution d'un problème de planification utilisé par la NASA pour quantifier les performances de tels outils et des machines qui les supportent ("Monkey and bananas", 30 règles) a permis d'établir le classement suivant:

OUTIL (version) TOOL	MACHINE (mémoire) (memory)	Nb règles déclenchées Fired rules	Temps en secondes Time(s)	Nb règles par sec. Fired rules/s
SPOCK-C (V1)	SUN (RISC) 4-260	82	0.03	2733
SPOCK-C (V1)	SUN 3-160(4Mo)	82	0.14	586
SPOCK-Lisp (V1)	SUN 3-160(4Mo)	82	0.18	455
SPOCK-Flavors (V1)	SYMBOLICS 3620(8Mo)	82	0.31	259
ART (V2)	SYMBOLICS 3640(4Mo)	86	1.2	72
OPS-V (DEC V2)	VAX 11/780	81	1.3	62
OPS-V (Forgy V2)	SYMBOLICS 3640(4Mo)	81	1.7	48
ART (V2)	TI Explorer (4Mo)	86	2.4	36

La programmation des règles pour les autres outils que SPOCK a été réalisée par la NASA.

Commentaires :

Le rapport de performance entre SPOCK et OPS-V a deux raisons:

- le réseau de filtrage est représenté dans OPS-V sous forme d'un micro-programme interprété, tandis que pour SPOCK le parcours du réseau est totalement compilé.

- le traitement des suppressions et des modifications d'objets est, dans SPOCK,une variante originale de l'algorithme de RETE, avec de plus la modification des objets "en place", c'est à dire sans copie, contrairement à OPS-V.

Ces mêmes raisons expliquent une partie de l'avantage de SPOCK sur ART ; une autre part est attribuable à l'allègement de certains traitements dans SPOCK par rapport à ART (pas de test d'unicité à chaque insertion de fait, par exemple) ; en contre-partie, certaines solutions accélératrices opérationnelles dans ART (recherche en mémoires locales par hachage dynamique sur les tests d'égalité, par exemple) ne sont pas encore implantées dans SPOCK.

Les temps de SPOCK-Lisp dépendent de la représentation des objets sur lesquels s'appliquent les règles. Sur SYMBOLICS, de meilleures performances auraient pu être obtenues en utilisant des "defstruct" plutôt que des "Flavors".

La similarité des performances de SPOCK-Lisp et SPOCK-C dans ce test est due à l'absence d'opérations arithmétiques qui, en LISP, demanderaient des tests de type à l'exécution.

3.1 CYCLE D'INFERENCE DE SPOCK

Le cycle d'inférence de SPOCK comporte quatre phases :

- une phase de filtrage (1) ("pattern matching"), qui détermine, à partir de l'ensemble courant des objets ("working memory") et de l'ensemble fixe des règles, l'ensemble des règles instanciées, c'est à dire des uplets de la forme : (règle, objet1, ..., objetN) formés d'un nom de règle et d'une combinaison des objets qui satisfait le membre gauche de la règle.

- une phase de sélection (2) ("conflict resolution"), qui détermine l'instance de règle de plus forte priorité ; en cas de priorités égales, le choix est aléatoire.

- une phase de déclenchement (3), qui consiste à exécuter la partie droite de la règle sélectionnée sur les objets de l'instance, ce qui a pour effet de modifier l'ensemble des objets et/ou de communiquer des actions au monde extérieur.

- une phase de communication (4), qui consiste à exécuter la fonction communication ; cette fonction, qui par défaut ne fait rien, est à redéfinir par l'utilisateur, par exemple pour asservir un objet horloge à une horloge externe, prendre en compte des modifications externes (acquisition de données en temps réel), gérer un échéancier,...

Fonctionnement du cycle d'inférence :

SPOCK calcule l'ensemble des règles instanciées de façon incrémentale, à chaque cycle d'inférence. La phase 1 de filtrage correspond donc à des calculs de mise à jour de l'ensemble des conflits, qui ont lieu en réalité après chaque modification de l'ensemble des objets, c'est à dire pendant les phases 3 et 4.

Dans l'état initial, l'ensemble des objets et l'ensemble des règles instanciées sont vides. A l'initialisation, plusieurs objets sont insérés dans la mémoire de travail, ce qui peut provoquer l'instanciation de plusieurs règles qui sont placées dans l'agenda. Au démarrage du cycle d'inférence, l'instance la plus prioritaire est déclenchée : ces actions modifient éventuellement la mémoire de travail-et conséquemment l'agenda, par filtrage-, et le cycle se poursuit jusqu'à ce que l'une des conditions suivantes se présente :

- à l'étape de sélection, l'agenda est vide : il y a arrêt du cycle ;
- pendant les phases 3 ou 4, la fonction "halt" est appelée : il y aura arrêt du cycle à la phase de sélection suivante ;
- pendant l'une quelconque des phases, une interruption se produit (break ou erreur) : il y a arrêt immédiat du cycle.

Dans ce dernier cas, le redémarrage du cycle est possible par la fonction "restart", à condition de restaurer un état cohérent par l'appel de la fonction "clearandmatch".

Dans le cas d'applications de systèmes experts temps réel, il convient d'inclure le cycle d'inférence dans une boucle plus générale d'attente ou de gestion d'informations extérieures, qui permette de relancer automatiquement le cycle d'inférence.

3.2 LES OBJETS GERES PAR SPOCK

Les objets sur lesquels portent les règles sont des structures munies d'opérations abstraites pour la création, le test de type, l'inspection et l'affectation des attributs. En LISP, les objets peuvent être représentés par n'importe quelle structure définie par "defstruct", ou une extension par des objets munis de valeurs actives, attachement procédural, héritage multiple, ..., comme par exemple les Flavors.

Les règles sont génériques et fonctionnent indépendamment de toutes les caractéristiques des objets de l'application.

Dans les versions 2 et 3 de SPOCK, les assertions logiques peuvent être représentées à l'aide de faits vérifiant un test d'unicité ; les faits sont des objets internes, qui, à l'inverse des objets abstraits, ne peuvent pas être partagés avec le reste de l'application.

Un exemple d'objets est donné au chapitre 4.2.2.

3.3 LES REGLES GEREES PAR SPOCK

Leur complexité variera avec les versions successives de SPOCK.

La version 1 de SPOCK comporte des règles de type chaînage avant. Les règles possèdent un nom, une priorité, une partie gauche, une partie droite.

La partie gauche exprime une condition sur la mémoire de travail (dans un langage de type logique du premier ordre).

La partie droite exprime les actions à entreprendre lorsque la partie gauche est vérifiée.

Une règle s'apparente à une implication logique de la forme:

$\mu\ o_1{:}t_1 \dots \mu_k\ o_k{:}t_k \rightarrow$ actions

dans laquelle les objets o_i de type t_i sont quantifiés par μ_i -, universellement, existentiellement ou négativement.

Les types des objets sont exprimés par des schémas ("patterns")

Des exemples de règles sont donnés au chapitre 4.2.3.

La version 2 de SPOCK offre de plus un système de maintien des déductions (Truth Maintenance System) et de structuration de la base de règles en paquets.

La version 3 offrira des fonctionnalités pour structurer l'ensemble des faits en une hiérarchie de contextes et explorer en parallèle plusieurs hypothèses.

3.3.1 LES MEMBRES GAUCHES DES REGLES.

Le membre gauche d'une règle est une conjonction de conditions.

Une condition peut être:

- un schéma quantifié universellement, existentiellement ou négativement,
- un appel à LISP correspondant à une condition satisfaite si le résultat est différent de nil, fausse sinon,
- une liaison de variable à la valeur d'une expression LISP, correspondant à une condition toujours satisfaite (afin de mémoriser le résultat d'un calcul intermédiaire).

Les schémas sont par défaut universellement quantifiés.

La quantification existentielle est repérée syntaxiquement par le mot clé exists devant le schéma.

La négation est repérée par le mot clé not.

Exemple :

```
 ConditionsList :
                    condition
                    ConditionsList condition;
condition:          pattern
                    symbol':'pattern
                    EXISTS pattern
                    NOT pattern
                    STRING
                    symbol'='STRING;
```

Un schéma exprime un type d'objet sous la forme d'une conjonction de tests.

Le premier test concerne la structure à laquelle appartient l'objet. Ce test est compilé avec le prédicat typep de LISP, qui est vrai si l'objet est une instance de la structure ou d'une extension de la structure.

Les autres tests portent sur les champs des objets, ou sur l'objet tout entier.

Les tests sur un champ suivent le nom du champ ; ils sont formés :

- soit d'un prédicat prédéfini et d'une valeur (constante, variable ou expression LISP),
- soit d'un appel à LISP de la forme 'predicat arg2...argN qui sera compilé en (predicat arg1 arg2...argN) où arg1 est la valeur du champ.

Les tests sur l'objet tout entier sont possibles dans les appels à LISP repérés par le mot call à l'intérieur du schéma. Il suffit pour cela de faire apparaître dans l'expression LISP la variable self qui est liée à l'objet candidat au schéma.

3.3.2 CONVENTION SUR L'ORDRE DES TESTS

Le compilateur de règles ne change pas l'ordre des prémisses dans les règles, ni l'ordre des tests dans les schémas.

Le compilateur partage tous les noeuds du réseau qu'il est possible de rendre communs à plusieurs règles sans perturber l'ordre des tests.

3.3.3 LES MEMBRES DROITS DES REGLES

Ils représentent des suites d'actions à exécuter dans l'environnement des variables liées par le membre gauche. Une action peut être:

- une expression LISP quelconque ;
- une liaison par bind de variables intermédiaires à des valeurs quelconques ;
- une action prédéfinie de modification de la mémoire de travail :

  * insertion d'un nouvel objet

  * suppression d'un objet

  * modification d'un objet,avec et sans filtrage

3.3.4 LES PRIORITES DANS LES REGLES

Les priorités sont des valeurs numériques entières ou réelles.

La priorité d'une règle peut être statique ou dynamique.

Les priorités dynamiques sont définies par :

- la valeur d'une variable liée dans le membre gauche,
- une expression LISP quelconque, pouvant porter sur les variables liées du membre gauche.

Les priorités dynamiques sont évaluées pendant la phase de filtrage, dès qu'une instance de règle est trouvée.

Les priorités statiques sont définies par une constante entière ou prédéfinie (**maximum**, **minimum**, **immediate-firing**).

La constante **immediate-firing** indique que la règle est déclenchée dès qu'une instance est trouvée. Les règles employant cette priorité doivent respecter quelques précautions d'emploi et être réservées pour des actions simples.

4 APPLICATION DE SPOCK : ASSISTANT TACTIQUE

Les travaux en cours doivent permettre d'obtenir dans un premier temps un outil d'assistance à la préparation de missions de type reconnaissance avec un hélicoptère léger, puis une aide à la navigation dans un champ de menaces, tenant compte du relief grâce à une base de données cartographique embarquée.

L'expertise est recueillie auprès d'un Chef Pilote de l'Aviation Légère de l'Armée de Terre (ALAT).

Les problèmes à gérer sont:

- la prise en compte de l'intervisibilité entre l'hélicoptère et les éventuelles menaces (avec meilleure utilisation possible du relief).
- l'intervisibilité hélicoptère-but (cas du tir).
- la génération de trajectoires
- le pilotage de la génération de trajectoires tenant compte des menaces, du relief, du respect de la mission (heure, but).

Pour démontrer l'intérêt de SPOCK pour ce type de problèmes, un simulateur maquette basé sur une description toponymique d'une zone géographique a été réalisé sur machine SYMBOLICS.

4.1 SIMULATEUR POUR L'ASSISTANT TACTIQUE

4.1.1 PARAMETRES D'ENTREE

L'utilisateur définit :

- la position de départ de l'hélicoptère
- la vitesse de l'hélicoptère
- la destination finale
- les différentes menaces, symbolisées par :
    * leur nom
    * leur position
    * leur portée
- les règles d'évolution de l'environnement :
    * le maintien des objectifs (généralement atteindre la destination)
    * la cohérence des prises de décision avec l'environnement (éviter les menaces, par exemple).

4.2 METHODE DE SIMULATION

4.2.1 PRISE EN COMPTE DU CARACTERE TEMPOREL

La simulation du temps s'opère par la gestion d'un objet de type Horloge qui comporte l'information Temps. Le Temps s'incrémente par une règle de priorité minimum quand tous les objets du système ont été traités pour l'instant de Temps courant.

Le mouvement des objets mobiles s'opère de la même manière par une règle de priorité minimum, quand l'objet se trouve être en retard sur la variable Temps. Grâce à cette priorité minimale, le mouvement n'est effectué que lorsque tous les paramètres d'ajustement du déplacement (vitesse et cap) ont été positionnés.

4.2.2 STRUCTURE DES OBJETS

La simulation met en oeuvre quatre types d'objets (Flavors) :

- Hélicoptère
- Destination
- Obstacle(s)
- Horloge

Ces objets sont construits à partir d'héritage des objets de base suivants :

```
- INTITULE  : champ NOM

- HORLOGE   : champ TEMPS

- DIMENSION : champ PORTEE

- POSITION  :

      * champ X  )
                 ( position dans le plan horizontal
      * champ Y  )

      * champ Z    altitude

- VITESSE :

      * champ Vx  )
                  ( variation de position horizontale pour
                  ( chaque mouvement
      * champ Vy  )

      * champ Vz   variation d'altitude pour chaque mouvement
```

Les objets définitifs sont construits par héritage :

- OBJET hérite de :
    - * INTITULE
    - * POSITION
- MOBILITE hérite de :
    - * VITESSE
    - * HORLOGE
- OBJET-MOBILE hérite de :
    - * OBJET
    - * MOBILITE
- HELICOPTERE hérite de OBJET-MOBILE
- DESTINATION hérite de OBJET
- OBSTACLE hérite de :
    - * OBJET
    - * DIMENSION

REMARQUE : Les menaces réelles seraient de type OBSTACLE-MOBILE.

4.2.3 REGLES D'EVOLUTION DE LA SITUATION

Les règles sont mises en forme par la macrocommande DEFRULE qui prend comme arguments :

- le Nom de la règle (réutilisable par les outils de trace et de contrôle d'exécution du programme)
- la Priorité de la règle (utilisée pour le classement des déclenchements des règles dans l'agenda)
- le Corps de la règle : prémisses puis conclusions, reliées par le symbole "→"

Règle d'incrémentation du temps :

Cette règle ne prend en compte qu'un objet : l'HORLOGE.

```
(DEFRULE INCREMENTATION 0
    H:=(HORLOGE TEMPS=Tps)
    →
    (MODIFY H TEMPS =(+1 Tps))
```

Règle de déplacement des objets

```
(DEFRULE MOUVEMENT 0
    O:=(OBJET-MOBILE
           X=x
           Y=y
           Z=z
           Vx=vx
           Vy=vy
           Vz=vz
           TEMPS=tom)
   (HORLOGE
           TEMPS=Tps)
   (>Tps tom)   :l'objet est en retard sur le temps
   →
   (MODIFY O X=(+x vx)
            Y=(+y vy)
            Z=(+z vz)))
```

Règle de navigation vers l'objectif (destination)

Pour s'assurer que l'hélicoptère va atteindre l'objectif, on compare son cap avec l'azimut de la droite joignant sa position présente et celle de l'objectif ; une différence trop importante se traduira par une modification de cap pour ramener la déviation à une valeur tolérable.

```
(DEFRULE VERS-DESTINATION 3
     H:=(HELICOPTERE
            X=x
            Y=y
            Vx=vx
            Vy=vy)
     D:=(DESTINATION
            X=dx
            Y=dy)
     (>(-(ATAN vy vx)(ATAN (-dy y)(-dx x)) )0.1)
     →
     (MODIFY H X=(*(COS(ATAN (-dy y)(-dx x)))
                   (SQRT(+(* x x)(* y y)))
                Y=(*(SIN(ATAN (-dy y)(-dx x)))
                   (SQRT(+(* x x)(* y y)))))
```

Règle d'évitement des obstacles(menaces)

Pour éviter un obstacle (menace), on considère en premier lieu ceux qui se trouvent sur la trajectoire de l'hélicoptère ; l'évitement correspond à *une modification de cap*.

La sélection peut se faire sur deux critères principaux :

- l'hélicoptère se trouve dans la portée de la menace ;
- l'hélicoptère se trouvera dans la portée de la menace dans k unités de Temps s'il ne change pas de cap (ni de vitesse ou d'altitude).

Le premier critère correspond à l'apparition d'une menace, soit que cette dernière soit mobile, soit qu'un changement du relief conduise à une intervisibilité hélicoptère-menace ; ce critère n'est pas encore pris en compte dans le simulateur, qui utilise le deuxième critère avec k=3.

Différentes stratégies de correction de trajectoire peuvent être mises en oeuvre ; il a été décidé de prendre comme nouveau cap la direction de la tangente au cercle de portée de la menace, au point où la trajectoire de l'hélicoptère aurait du couper ce cercle de portée de la menace. La règle s'écrit donc :

```
(DEFRULE EVITE-OBSTACLE 2
    H:=(HELICOPTERE
            X=x
            Y=y
            Vx=vx
            Vy=vy)
    O:=(OBSTACLE
            X=ox
            Y=oy
            PORTEE=p)
    (<=(+(EXPT (- ox (+ x(* 2 vx)))2)
          (EXPT (- oy (+ y(* 2 vy)))2))
        (EXPT p 2))
    →
    (MODIFY H
        Vx=(* (SQRT (+(EXPT vx 2)(EXPT vy 2)))
                (COS (+(ATAN (-oy y)(-ox x))

                        (*(SIGNUM(-(ATAN (-oy y)(-ox x))
                                    (ATAN (vy vx)))
                          (/ PI -2)))))
        Vy=(* (SQRT (+(EXPT vx 2)(EXPT vy 2)))
                (SIN (+(ATAN (-oy y)(-ox x))
                        (*(SIGNUM(-(ATAN (-oy y)(-ox x))
                                    (ATAN (vy vx)))
                          (/ PI -2))))))
```

4.3 PERFORMANCES

Pour un nombre d'obstacles restreint (moins de trente), on obtient des performances comprises entre 200 et 250 règles déclenchées par seconde, ce qui est parfaitement cohérent avec les résultats obtenus avec le benchmark NASA (cf tableau).

4.4 AMELIORATIONS PREVUES

La partie Simulateur prendra en compte la notion de volume de menace et se raccordera à l'équipement de base de données cartographique afin de prendre en compte le relief tant pour la navigation que pour l'évitement des menaces.

REFERENCES

[AF88]
Albert, L.,Fages,F.,Average Case Complexity Analysis of the Rete Multi-Pattern Match Algorithm.INRIA Research Report.Submitted to ICALP'88 Finland.July 1988

[BP 84]
Bruynooghe,M. and Pereira,L.M,Deduction revision by intelligent backtracking,in Implementation of Prolog,Campbell ed.,Ellis Horwood.1984

[CCF 88]
Codognet,C.,Codognet,P.,Filé,G.,Depth-first Intelligent Backtracking, Séminaire sur la Programmation en Logique,Trégastel,Mai 1988.Submitted to the International Conference on Logic Programming,Seattle,1988.

[Cla 85]
Clayton,B.D., ART Reference Manual,Inference Corporation.Los Angeles, California April.1985

[Fag 86]
Fages,F.,On the Proceduralization of Rules in Expert Systems,First France-Japan Symposium on Artificial Intelligence,Tokyo,Nov.86.In Programming of Future Generation Computers,Addison-Wesley,Eds.M. Nivat and K.Fuchi.

[For 79]
Forgy,C.L., On the Efficient Implementation of Production Systems. Department of CoSci,Carnegie Mellon University.1979.

[For 81]
Forgy,C.L.,OPS5 Users Manual,Department of CoSci,Carnegie Mellon University.1981.

[For 82]
Forgy,C.L.,Rete:A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem,Artificial Intelligence 19,pp.17-37.1982.

[For 84]
Forgy,C.L.,The OPS83 Report,Department of CoSci,Carnegie Mellon University.May 84.

[GD 87]
Ghallab,M.,Dufresne,P.,Moteurs d'inférences pour systèmes de règles de production:techniques de compilation et d'interprétation,LAAS Toulouse.1987.

[Gup 84]
Gupta,A.,Parallelism in Production Systems:the Sources and the Expected Speed-up,Depr. of Computer Science,Carnegie Mellon University,Pittsburgh. December 1984.

[deK 84]
de Kleer, J.,Choices Without Backtracking.Intelligent Systems Laboratory XEROX Palo Alto California,in Proceedings Fourth National Conference on Artificial Intelligence Austin Texas,pp.79-85.1984.

[deK 86]
de Kleer, J.,An Assumption-based TMS,in Artificial Intelligence vol 28-2,pp. 127-162,pp. 163-196.March 1986.

[LN 86]
Lee, N.S., Nejmeh,B.A., Towards an Expert-System Development Environment for Mixed Knowledge Representations and Programming Methodologies.Proc. of the 6th Workshop on Expert Systems and their Applications.Avignon.1986.

[Mir 87]
Miranker,D.P., TREAT:A Better Match Algorithm for AI Production Systems,Proceedings of the 1987 National Conference on Artificial Intelli-gence.Seattle,Washington.1987.

[Proc 84]
Proceedings of the Third Seminar on Application of Machine Intelligence to Defense Systems,Royal Signals and Radar Establishment.June 1984.

[Ril 86]
Riley, G.D., Timing Tests of Expert Systems Building Tools,Memorandum FM7(86-51),Artificial Intelligence Section,Lyndon B. Johnson Space Center, NASA.April 1986.

[Ril 87]
Riley,G.D.,private communication Mars 1987.

[SBMC 83]
Stefik,M., Bobrow,D.,Mittal,S.,Conway,L.,Knowledge Programming in LOOPS: report on an Experimental Course,The AI magazine,Fall 83,pp 3-13.1983.

[SF 88]
Schang, T., Fages, F., Real-Time Expert System for On-Board Radar Identification.55th Symposium AVP-AGARD on Software Engineering and its Applications to Avionics/April 88.

[War 85]
Warren,D.H.D.,An abstract Prolog Instruction Set.Technical Note 309, SRI International.1985.

[WGFC 86]
Wright, M.L., Green, M.W., Fiegl, G., Cross, P.F., An Expert System for Real-Time Control,SRI International,in IEEE Software.March 1986.

UTILISATION DE EMICAT POUR LE DEVELOPPEMENT DE SYSTEMES EXPERTS AEROSPATIAUX

Bruno MEHU, Antoine VIVANCOS
Electronique Serge DASSAULT
55, quai Marcel DASSAULT
92214 SAINT-CLOUD, FRANCE
tél : (1) 49 11 80 00

RESUME

Cet article présente les principales difficultés rencontrées lors de la réalisation en PROLOG de systèmes experts dans un contexte industriel. Il montre la nécessité d'une extension de PROLOG vers un langage orienté objet et décrit les mécanismes de base devant y figurer pour faciliter la programmation des systèmes experts. L'article décrit ensuite l'environnement construit sur cette extension pour permettre la représentation et l'exploitation de connaissances sous la forme de règles de production. La seconde partie de l'article présente quatre applications réalisées par l'ESD dans le domaine aéronautique et spatial : un système expert d'aide à la définition de liaisons électroniques dans un avion (BASILE), un système expert d'aide à la gestion des batteries utilisées sur satellite (GIBUS), deux systèmes experts d'aide au diagnostic de pannes de sous-systèmes satellite (SCAO : système de contrôle d'attitude et d'orbite, PCS : système pour l'alimentation électrique).

Mots clés :
environnement pour systèmes experts, systèmes experts, PROLOG, langage objet.

I. - LES BESOINS D'UN INDUSTRIEL

L'Electronique Serge DASSAULT (ESD) s'est intéressée très tôt aux techniques d'intelligence artificielle, pour résoudre des problèmes industriels sans solution classique satisfaisante (tels que le calcul symbolique, le diagnostic de pannes de circuits électroniques), et pour le maquettage de systèmes complexes. L'utilisation croissante de ces techniques dans de nombreuses applications a fait apparaître le besoin d'outils opérationnels dont les caractéristiques étaient les suivantes :

- multi-domaines : aide à la conception, "monitoring" et surveillance, diagnostic de panne, planification,...
- multi-techniques : logique des prédicats, systèmes à base de règles de production, logique floue, programmation objet, ...
- fiable,
- documenté,
- ouvert sur l'environnement logiciel existant,
- disponible sur les moyens informatiques de l'ESD pour faciliter la diffusion des applications au sein de l'entreprise, et dans un environnement PROLOG, le langage de base retenu pour nos développements en intelligence artificielle.

II. - COMMENT DEVELOPPER DES SYSTEMES EXPERTS EN PROLOG

Bien que la programmation des systèmes experts apparaisse souvent comme un domaine d'application privilégié du langage PROLOG, beaucoup de tentatives d'utilisation de ce langage dans ce domaine se sont soldées, sinon par un échec, tout au moins par beaucoup de désillusions. On a pu ainsi entendre dire que, bien qu'incorporant un moteur d'inférences, PROLOG offrait des mécanismes de raisonnement insuffisants, qu'il ne permettait pas l'écriture de méta-règles, qu'il n'était pas assez performant...

Ces points de vue contradictoires tiennent au fait que plusieurs approches sont envisageables en ce qui concerne l'utilisation de PROLOG pour le développement de systèmes experts ; nous allons les passer rapidement en revue.

- **PROLOG : langage de spécification**

  Dans cette approche, le système expert est programmé directement en PROLOG. Les connaissances sont représentées par des axiomes. Le raisonnement mis en oeuvre est celui fourni par le langage lui-même, c'est-à-dire le chaînage arrière avec recherche en profondeur d'abord. Il est clair que, si l'on s'y cantonne, cette façon de procéder possède de nombreux inconvénients qui, même au prix d'extensions telles que celles figurant dans APES [HAMMOND 84], limitent fortement les possibilités des systèmes experts ainsi développés :

  . pas de structuration de la connaissance,

  . stratégie fixe,

  . expression des méta-connaissances difficile,

  . absence de retour arrière intelligent...

- **PROLOG : langage d'implémentation**

  PROLOG est ici utilisé comme un langage général de programmation ; cela conduit à définir un formalisme de représentation et d'exploitation des connaissances (ou même tout simplement à reprendre un formalisme déjà existant tel que EMYCIN [MELLE 81], OPS5 [FORGY 81], SNARK [LAURIERE 83], TANGO [CORDIER 83]...) et à l'implémenter en PROLOG. C'est ce que l'on appelle un "shell". Cette méthode présente l'avantage de permettre une implémentation très rapide en raison de la puissance de PROLOG en tant que langage de programmation. Par contre, l'efficacité résultante n'est pas garantie et la représentation des connaissances ne peut s'effectuer qu'à travers le formalisme choisi ce qui est souvent insuffisant.

- **Extensions de PROLOG**

  Cette troisième voie consiste à prendre acte des insuffisances de PROLOG et à définir une extension du langage offrant es fonctionnalités nécessaires à la réalisation de systèmes experts. De nombreux travaux ont vu le jour sur ce thème, la plupart visant à introduire des possibilités de structuration de la connaissance et de contrôle du raisonnement [DINCBAS 83], [CLARK 82], [OGAWA 84]. Bien qu'aucun de ces travaux n'ait atteint le stade industriel, un certain nombre de principes ont été établis :

  . PROLOG doit rester un *sous-ensemble* du nouveau langage défini,

  . les extensions doivent avoir une sémantique parfaitement définie,

  . les extensions doivent offrir des mécanismes puissants (supérieurs au simple ajout/retrait de clauses) permettant la définition et la modification de bases de connaissances.

Confrontés au problème de la réalisation de systèmes experts dans un contexte opérationnel, nous avons développé l'environnement EMICAT [TAILLIBERT 86] en nous appuyant sur la troisième approche (extensions de PROLOG) et en essayant de mettre l'accent sur les fonctionnalités nécessaires au développement de systèmes experts dans un contexte industriel à savoir :

- possibilité d'implémenter une large gamme de formalismes de représentation de connaissances,

- facilités pour la définition de stratégies propres aux problèmes à traiter,

- réalisation rapide et peu coûteuse de prototypes de systèmes experts.

III. - EMICAT

EMICAT est une extension de PROLOG vers un langage orienté objet (LOO) sur lequel a été développé un environnement pour l'exploitation de règles de production.

3.1. Le langage orienté objet

Le LOO est basé sur le concept de "frame" [MINSKY 75] (et plus précisément sur le modèle FRL [ROBERTS 77]) auquel a été ajouté un mécanisme de communication entre objets par messages. Le concept d'objet s'est avéré un concept à la fois puissant et efficace pour la présentation des connaissances factuelles ou des raisonnements de bas niveau dans les systèmes experts [FARGUES 83].

Par ailleurs, les possibilités d'attachements procéduraux (ou réflexes) ont été étendues afin, en particulier, de permettre une spécification plus précise de l'instant de déclenchement. Ainsi, à tout attribut, il est possible d'attacher les procédures suivantes : si-besoin, avant-ajout, après-ajout, avant-retrait, après-retrait, avant-modification, après-modification.

Enfin, certaines facilités ont été introduites parmi lesquelles :

- la spécification de la portée de l'héritage (afin de gagner en efficacité),

- un contrôle anti-bouclage au niveau des attachements procéduraux pour faciliter la programmation d'une stratégie de chaînage arrière adaptée au problème à traiter,

- une trace programmable pour l'aide à la mise au point de la base de connaissances,

- un mécanisme de mémorisation d'état qui fait l'objet du paragraphe suivant,

- des primitives graphiques pour le développement de dialogues homme-machine conviviaux,

- un éditeur d'objets.

Mémorisation d'état

Un outil de développement de systèmes experts doit offrir les trois mécanismes de choix suivants [LAURENT 84] :

- choix de l'état sur lequel agir,

- choix de la règle à appliquer,

- choix des objets sur lesquels appliquer la règle.

Le premier de ces choix est le plus coûteux à mettre en oeuvre car il nécessite une mémorisation d'information souvent importante ; il est significatif que peu d'outils de développement le proposent.

Il apparaît que si ce mécanisme est intégré au LOO, le concepteur de système expert dispose alors d'un outil élémentaire lui permettant la programmation de stratégies adaptées au problème à traiter. Pour cela, dans EMICAT, un prototype - nommé "état" - est prédéfini ; à tout moment, les opérations effectuées sur les objets sont mémorisées dans l'instance de ce prototype désignée comme "état courant". A chaque fois que l'on souhaite créer un point de retour, une nouvelle instance de l'objet "état-courant" doit être engendrée. Il est ainsi possible de mémoriser tout ou partie de l'arbre de recherche, et ensuite de s'y déplacer.

Les informations mémorisées dans chaque objet et nécessaires au changement d'état sont accessibles à l'utilisateur et permettent de programmer une stratégie où par exemple, en cas d'impasse, le système retourne à l'état ayant engendré un attribut donné, ainsi qu'on le trouve dans [VIALATTE 85].

Un système expert n'ayant pas nécessairement à recourir au choix d'état dans toutes les phases du raisonnement, ce mécanisme peut être mis hors service afin de ne pas pénaliser l'ensemble des recherches.

3.2. L'environnement pour règles de production

Un LOO ne suffit pas à réaliser un système expert car beaucoup de connaissances se représentent plus aisément sous la forme de règles de production. C'est pourquoi EMICAT comprend, construit sur le couple PROLOG-LOO, un environnement permettant la définition et l'exploitation de règles de production.

Les règles avec EMICAT

La définition d'un formalisme de règles de production est soumise à un ensemble de contraintes bien souvent antagonistes ainsi :

- le formalisme doit être puissant et général mais doit pouvoir également s'adapter à une expertise particulière pour permettre, dans certains cas, l'introduction des règles par l'expert lui-même,

- le formalisme doit permettre l'écriture de règles mais aussi de méta-règles, les premières traitant plutôt du domaine d'expertise et les secondes de stratégies de recherche,

- le formalisme doit être déclaratif mais doit permettre également l'expression de connaissances à caractère procédural (stratégies de recherche).

Pour tenir compte de ces contraintes, une règle EMICAT se représente sous la forme d'une conjonction de buts PROLOG.

Il est clair qu'un tel formalisme, s'il convient bien au cogniticien en lui offrant ainsi toute la puissance de PROLOG, ne facilite pas les échanges avec l'expert et en aucun cas ne lui permet de concevoir les règles lui même. C'est pourquoi la forme "conjonction de buts PROLOG" ne représente que la forme interne des règles. En pratique, l'utilisateur peut définir chaque règle comme un objet, héritant d'un prototype général appelé "règle", muni de deux attributs décrivant :

- l'un le "corps" de la règle sous la forme la mieux adaptée au problème,

- l'autre la "traduction" à appliquer au corps pour le transformer en une conjonction de buts PROLOG (forme interne).

Cette approche permet d'utiliser la relation hiérarchique entre les objets pour définir des familles de règles dont le formalisme est adapté au type de connaissances à représenter ; un même système expert pourra ainsi recourir à plusieurs formalismes simultanément.

La traduction s'effectue automatiquement à l'entrée ou à la modification de la règle par le jeu de réflexes prédéfinis hérités du prototype "règle". Grâce à l'utilisation des "opérateurs" présents dans la plupart des implémentations de PROLOG cette traduction peut rester simple tout en offrant à l'expert un formalisme facile à manipuler.

Comment "parler" des règles

L'importance des méta-connaissances n'est plus à démontrer [PITRAT 82] ; en effet dès qu'un système expert atteint le millier de règles, les mécanismes d'application qui, quels que soient les adressages associatifs mis en place, restent essentiellement séquentiels, finissent par perdre toute efficacité.
Pour aider à la détermination des connaissances utiles à la résolution d'un problème, EMICAT offre cinq possibilités pour se référer aux règles présentes dans la base :

- organiser les règles sous forme hiérarchique, ce qui est immédiat grâce à l. représentation objet sur laquelle elles sont construites ;

- construire un ou plusieurs index sur une classe de règles, la construction est faite automatiquement au moment de l'entrée de la règle, à partir de sa forme interne ;

- se référer à un ensemble d'attributs, calculés à l'entrée de la règle sur sa forme interne, repérant les constantes qui s'y trouvent mentionnées (objets, types, attributs, prédicat externe), de manière comparable à ce qui est fait dans [DAVIS 80] ;

- se référer à des attributs placés par le rédacteur de la règle (par exemple pour indiquer le coût ou l'importance de la règle),

- effectuer une unification ou un filtrage sur le corps de la règle (forme externe).

Application des règles

L'objectif principal de EMICAT était de permettre la définition de stratégies adaptées au problème à résoudre ; cela est rendu possible grâce à la primitive (unique) d'application des règles décrite au paragraphe ci-après. Celle-ci est utilisée aussi bien par les stratégies prédéfinies que par les stratégies définies par l'utilisateur.

Primitive d'application

C'est un prédicat qui comporte trois arguments et s'écrit :

appliquer (choix-de-règle, choix-d'objet, objet-sélectionné)

Chaque solution correspond à une instanciation réussie de l'une des règles choisies. Les différentes solutions sont obtenues, comme c'est toujours le cas en PROLOG, à chaque retour arrière sur la primitive. La signification des trois arguments est la suivante :

- le choix-de-règle s'effectue en donnant soit la liste des règles à appliquer, soit une propriété devant être vérifiée par celles-ci,
- le choix-d'objet est une liste de filtres permettant de spécifier, au moment de l'application, les objets sur lesquels la règle doit s'appliquer.
- L'objet-sélectionné permet une communication entre la règle et le mécanisme d'application.
  Ainsi, pour "appliquer les règles, sélectionnés par les méta-règles m1 et m2", il suffira d'écrire :

appliquer (R :: appliquer (m1.m2.nil, nil,R),nil,*)

Stratégies prédéfinies

EMICAT offre quelques stratégies prédéfinies construites à partir de la primitive appliquer ; parmi celles-ci on peut citer :

Passage	évaluation de toutes les solutions
Saturer	évaluations successives de "passage" jusqu'à ce qu'aucun objet ne soit plus modifié au cours d'un passage
Prouver	application jusqu'à saturation ou démonstration d'un but (conjonction de buts PROLOG) fourni en argument

Stratégies utilisateur

Ce n'est ni par hasard, ni par oubli, si nous n'avons pas encore évoqué le chaînage arrière. En effet, celui-ci ne doit pas être considéré comme un mode d'application des règles (celles-ci ne pouvant en toute rigueur s'appliquer que dans le sens condition - action) mais comme une stratégie de recherche. D'autre part, l'expérience acquise par la réalisation de plusieurs systèmes experts nous a montré qu'il n'était pas réaliste de proposer un chaînage arrière "standard" du fait du nombre élevé de choix possibles tels que :

- la recherche doit elle se faire en profondeur ou en largeur d'abord ? (ou en un mélange des deux),
- doit-on déclencher la recherche pour tous les objets ? (pour certains cela n'a souvent aucun sens),
- quand et pour quels objets doit-on poser des questions à l'opérateur ?

C'est pourquoi il nous semble que la définition de la stratégie de recherche en arrière est du ressort du cogniticien. EMICAT offre de nombreuses facilités pour implémenter une telle stratégie ; ainsi, par exemple, pour réaliser la démonstration d'un but de la forme : recherche de la valeur de l'attribut d'un objet donné, en profondeur d'abord, sans question utilisateur.

il suffit :

- de prévoir la génération d'un index sur les règles concluant à cette valeur,
- d'attacher à l'objet pour lesquels le chaînage arrière est souhaité un réflexe si-besoin qui applique les règles obtenues par cet index.

et c'est tout ! La recherche de la valeur de l'attribut déclenchera automatiquement la stratégie en chaînage arrière ; le système anti-bouclage intégré à la représentation objet évitant une éventuelle recherche infinie.

IV. - APPLICATIONS AERONAUTIQUES ET SPATIALES

Dans ce chapitre sont présentées quatre applications de EMICAT dans le domaine aéronautique et spatial.

4.1. Aide à la définition des liaisons électroniques (BASILE)

Une première version de BASILE a été réalisé conjointement par l'ESD et la société AMD-BA, pour la définition des connexions entre équipements du Système de Navigation et d'Armement d'avions de combat.

Fonctionnalités

Les connexions entre les équipements d'un *système d'armes* moderne doivent respecter des normes techniques qui sont à la fois nombreuses et complexes. D'autre part, les modifications du système amènent à relier des équipements déjà utilisés à de nouveaux équipements dont il faut spécifier l'interface. Le système BASILE apporte aux concepteurs une aide à la définition de liaisons entre équipements, en mettant à leur disposition la connaissance des techniciens spécialisés en liaisons électroniques. De plus, BASILE offre à l'expert la possibilité de modifier et d'améliorer, sans intervention d'un informaticien, la base de connaissances : la majeure partie des connaissances peut ainsi être saisie et validée directement par l'expert.

Caractéristiques

BASILE est un système interactif où l'utilisateur donne des valeurs de caractéristiques désirées, ou des contraintes sur ces valeurs. Le raisonnement guidé par les données propage alors toutes les conséquences de ces données en éliminant les solutions incompatibles. Ce processus incrémental permet d'aboutir à une liaison complètement spécifiée. Un mécanisme de retour arrière intelligent permet la modification partielle de la *solution en cours de construction*.

```
si niveau_logique de etat_logique_actif de emetteur est haut
alors tension de etat_logique_actif de emetteur doit_etre > = 24

si E est_un etat_logique
et niveau de E est isole
alors impédance de E doit_etre  >100 000
```

4.2. Gestion de batteries de sattelite (GIBUS)

Ce système est réalisé par l'ESD (*sous-traitants* : SAFT et COGNITECH) dans le cadre d'un contrat de l'Agence Spatiale Européenne (ESA).

Fonctionnalités

La gestion de batteries des sattelites en orbite basse est une tâche délicate qui nécessite une expertise spécialisée dans les centres de contrôle. En effet, les batteries d'accumulateurs spatiaux sont des équipements pour lesquels on ne dispose pas de modélisation mathématique du comportement, suffisamment générale pour être applicable dans tous les régimes de fonctionnement.

La gestion de batteries a deux objectifs :

1) effectuer à chaque orbite (durée 100 min) une recharge suffisante des accumulateurs pendant la période d'éclairement (65 min) afin de disposer de toute la *puissance nécessaire* à la plateforme et aux charges utiles pendant la période d'éclipse (35 min),

2) maximiser la durée de vie de la batterie, en évitant les surcharges excessives et en prenant en compte le vieillissement des accumulateurs dans l'ajustement des paramètres de gestion.

Les fonctions du système expert sont :

- **Le "monitoring" des mesures de chaque orbite**

  Le système intègre à la fin de chaque orbite un certain nombre de données caractéristiques et les qualifie : aberrante, nominale, anormale. En régime stationnaire (utilisation et recharge régulières de la batterie) le système élabore les valeurs attendues à l'aide de modèles expérimentaux. En régime transitoire (cas général) le système fait un bilan d'utilisation et de recharge sur un historique d'une vingtaine d'orbites. Le cas échéant un diagnostic d'anomalie du comportement de gestion de la recharge est produit ainsi que des conseils pour corriger ou compenser l'anomalie.

- **L'analyse à long terme de l'historique de la batterie**

  La fonction de "monitoring" construit incrémentalement un historique synthétique des données et événements importants. La fonction d'analyse fournit un langage de requêtes de cet historique ainsi que des *fonctions de diagnostic* à long terme concernant l'estimation de la durée de vie restante et les dégradations de performances.

Caractéristiques

Le système expert traite cycliquement les données de l'orbite courante. Le raisonnement utilise les données des orbites précédentes. Il est donc nécessaire de représenter dans la base de connaissances :

- des faits datés (mesures, conclusions des orbites courantes et passées),

- des faits instantanés (vrais pendant une orbite) et des faits ayant une portée temporelle (vrais pendant plusieurs orbites).

Des primitives de gestion de ces faits ont été réalisées aisément avec le LOO de EMICAT. Les faits peuvent être utilisés par les différentes familles de règles du système.

```
if end_of_charge_current is abnormal(sup)
and k_factor is nominal
then partial_short_circuit detected_during charge

if temperature  < 10
and abs(temperature - avg(temperature, m-5, m-1))> 2
then state is transient
```

4.3. Diagnostic de pannes du sous-système alimentation (PCS)

Ce système expert a été réalisé conjointement par l'ESD et l'Aérospatiale/DSSS (Cannes) dans le cadre d'un contrat CNES.

Fonctionnalités

Le sous-système PCS du satellite TDF1, régule l'alimentation électrique du satellite. Son fonctionnement est donc de nature purement électrique, une anomalie y correspond à un niveau incorrect de tension ou de courant sur une partie du sous-système ou d'un élément client alimenté.

Les missions demandées au système expert sont au nombre de trois :

- la détection d'anomalies, c'est-à-dire le contrôle fin et systématique de toutes les télémesures par comparaison à des valeurs nominales. Celles-ci sont calculées dynamiquement par modélisation en fonction de la configuration du satellite (détermination d'un état de référence).
- le diagnostic de pannes, c'est-à-dire la détermination des composants responsables des anomalies détectées et si possible de leurs modes de pannes.
- la saisie, la modification, la consultation des connaissances par un expert, non spécialiste en intelligence artificielle.

Caractéristiques

La détection d'anomalies nécessite la gestion d'un état de référence, c'est-à-dire les tensions et courants de toutes les liaisons des schémas électriques et donc les valeurs théoriques des télémesures. A chaque modification de la configuration, cet état de référence est recalculé. Les connaissances utilisées consistent en une description structurelle du sous-système, sous forme de hiérarchie de boîtes reliées par des liaisons, et une description fonctionnelle, sous forme de fonctions de transfert spécifiques à une boîte ou génériques pour un type de boîte (résistance par exemple). Ces fonctions relient les courants et tensions des liaisons en entrée et sortie de la boîte.

Pour les fusibles, la fonction de transfert est la suivante :

```
dans fusible on_a si état = passant alors i(2) := i(1)
```

Des règles de calcul de paramètres permettent de calculer les paramètres utilisés par les fonctions de transfert (puissance, résistance, état,...) en fonction des caractéristiques de la boîte ou de ses sous-composants.

Pour évaluer le paramètre "état" utilisé dans la fonction de transfert d'un fusible, on applique, par exemple, la règle de calcul de paramètre suivante :

```
si mode(fusible)=nominal
alors état(fusible) := passant
sinon état(fusible) := non_passant
```

Le calcul de l'état de référence est un algorithme de parcours du schéma exploitant des connaissances déclaratives. Les anomalies sont détectées par comparaison des valeurs de télémesures aux valeurs théoriques déduites de l'état de référence.

Pour le diagnostic, les connaissances utilisées consistent en un ensemble de règles attachées à chaque boîte, permettant de dédouaner (disculper) la boîte et tout son contenu, ou, au contraire, de faire des hypothèses de pannes à l'intérieur de la boîte. Ces règles s'appuient sur la normalité/anormalité des télémesures ou sur leurs valeurs numériques. Il s'agit donc d'un "compilation" de connaissances de modélisation et de connaissances des pannes possibles.

Le diagnostic consiste à parcourir, de façon descendante, le schéma en s'appuyant sur la description structurelle et à tenter de disculper ou suspecter chaque boîte. La réduction et corrélation des hypothèses de pannes réduites sont faites par calcul formel sur les expressions logiques.

```
bloc1 est_dédouané si tm1 ≈ 0.0
            ou tm2 ≈ puissance(bloc1)/max (20.0, tm1)

où "≈" signifie à peu près égal et où bloc1 est une boîte et tm1, tm2 sont des télémesures.
```

4.4. Diagnostic de pannes du sous-système de contrôle d'attitude et d'orbite (SCAO)

Ce système expert a été réalisé conjointement par l'ESD et Matra Espace (Toulouse) dans le cadre d'un contrat CNES.

Fonctionnalités

Le sous-système SCAO du satellite TELECOM1, assure le positionnement et la stabilisation du satellite en orbite géostationnaire. Son fonctionnement est de nature à la fois mécanique (dynamique) et électronique, les deux aspects étant étroitement liés dans les boucles de pilotage (roulis, tangage).

Une anomalie dans le sous-système peut se traduire par un dysfonctionnement purement électrique, par exemple dans l'alimentation, ou par des phénomènes dynamiques indésirables ayant des aspects temporels et transitoires.

Les missions demandées au système expert sont au nombre de trois :

- le diagnostic d'anomalies : partant des anomalies détectées au centre de contrôle, il s'agit de localiser la panne à un niveau de finesse compatible avec les capacités de reconfiguration existantes à bord (ce qu'il est coutume d'appeler "l'élément commutable").

- la formation des opérateurs, par l'intermédiaire de facilités de visualisation et d'exploitation des raisonnements et d'accès aux connaissances.

- la saisie, la modification, la consultation des connaissances par un expert, non spécialiste en intelligence artificielle.

Caractéristiques

De même que dans le PCS, les connaissances relatives à ce sous-système peuvent être classées en trois catégories :

- les schémas du SCAO : leur description est analogue à celle du PCS,

- les connaissances fonctionnelles qui décrivent les dépendances fonctionnelles entre les divers constituants du sous-système, trois types de connaissances fonctionnelles sont fournies au système :

  1) **Des lois générales** portant sur la structure du sous-système, mais indépendantes du sous-système considéré. Par exemple, les règles d'influence électrique :

```
dans les relais2_1, on a
 si position = on
 alors
   entree(1) en mode M influence sortie(3) en mode M
 sinon
   entree(2) en mode M influence sortie(3) en mode M
```

  Cette règle est une règle générique, elle est valable pour tous les relais de type relais2_1. Elle exprime le fait que le mode en entrée se propage en sortie.

  2) **Des relations spécifiques** au sous-système, non déductibles de la modélisation structurelle. Elles portent sur les connaissances dynamiques en fournissant des relations impliquant des variables systèmes et les télémesures.

```
roulis_reconstitué en mode 3
 est_influencé_par delta_x_moins en mode anormal,
                   et moment_cinétique en mode anormal
```

  3) **Des cas d'exception**, qui permettent de prendre en compte des cas particuliers dans la structure du sous-système.

```
tc456 en mode on n'influence pas liaison1 en mode à_zéro.
```

- les connaissances heuristiques qui permettent d'orienter les recherches sur les éléments les plus soupçonnables (degré de suspicion, ...).

Le SCAO est découpé en sous-ensembles fonctionnels (dynamiques ou électriques). Un premier diagnostic est effectué sur le sous-ensemble du SCAO dans lequel a été détectée la première anomalie. Si la panne n'a pas été localisée dans ce premier sous-ensemble, de nouveaux diagnostics sont lancés sur les sous-ensembles qui ont une influence sur l'anomalie.

Le raisonnement à l'intérieur d'un sous-ensemble, consiste à identifier toutes les causes possibles de l'anomalie à partir du schéma, de la configuration et des dépendances fonctionnelles, puis à construire une séquence de tests sur les valeurs des télémesures permettant de discriminer les causes possibles. Les tests exploitent, en priorité, la liste des anomalies fournies au début du diagnostic. Par la suite, le système interroge l'opérateur sur la valeur des télémesures concernées par les tests.

V. CONCLUSION

Les systèmes experts présentés mettent en évidence, pour chaque applications, les spécificités des connaissances et également des raisonnements et stratégies utilisés. Il est bien évident par ailleurs que ce ne sont pas les mécanismes de base des règles de production (chaînage avant ou arrière) qui peuvent apporter une solution toute faite pour réaliser l'ensemble de ces systèmes.

Il faut donc disposer d'outils ouverts et puissants (de type EMICAT), offrant des facilités pour modéliser la base de connaissances et les stratégies désirées.

REFERENCES

[CLARK 82]	K.L. CLARK, F.G. Mc CABE, S. GREGORY, I.C. **PROLOG Language Features**, in LOGIC PROGRAMMING, Academic Press 1982
[CORDIER 83]	M.O. CORDIER, M.C. ROUSSET, **TANGO : Moteur d'inférences pour un Système Expert avec variables**, Rapport de recherche n° 123, LRI ORSAY, 1983
[DAVIS 80]	R. DAVIS, **Content reference : reasonning about rules**, Artificial Intelligence, Vol 15, n° 3, pages 223-239, Décembre 1980
[DINCBAS 83]	M. DINCBAS, **Contribution à l'étude des Systèmes Experts**, Thèse de Docteur Ingénieur, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, Janvier 1983
[FARGUES 83]	J. FARGUES, **Contribution à l'étude du raisonnement ; Application à la médecine d'urgence**, Thèse d'Etat Université PARIS VI, 1983
[FORGY 81]	C. FORGY, **The OPS5 user's manual**, CMU-CS81-135, Carnegie Mellon University, 1981
[HAMMOND 84]	P. HAMMOND, M. SERGOT, **APES : Augmented Prolog for Expert Systems**, Reference Manual, Micro-Prolog Version, Logic Based Systems Ltd, RICHMOND, SURREY, UK, 1984
[LAURENT 84]	Jean-Pierre LAURENT, **La structure de contrôle dans les Systèmes Experts**, TSI, Volume 3, n° 3, pages 161-177, 1984
[LAURIERE 83]	J.L. LAURIERE, **SNARK : Un moteur d'inférences pour Systèmes Experts en logique du premier ordre**, Rapport 430, Institut de programmation, Université de PARIS VI, 1983
[MELLE 81]	W. VAN MELLE, A. SCOTT, J. BENNETT, M. PEAIRS, **The EMYCIN manual**, Report n° STAN-CS-81-885, Department of Computer Science, STANFORD UNIVERSITY, 1981
[MINSKY 75]	M. MINSKY, **A framework for representing knowledge**, in P. WINSTON : the psychology of computer vision, New-York Mc GRAW-HILL, 1975
[OGAWA 84]	Y. OGAWA, K. SHIMA, T. SUGAWANA, S. TAKAGI, **Knowledge representation and inference environment : KRINE an approach to integration of frame, Prolog and graphics**, international conference on fifth generation computer systems, TOKYO, Novembre 1984
[PITRAT 82]	J. PITRAT, **Les connaissances déclaratives**, Colloque Intelligence Artificielle, Publication n° 30 du GR 22 - CNRS, LE MANS, Septembre 1982
[ROBERTS 77]	R.B. ROBERTS, I.P. GOLDSTEIN, **The "FRL" Primer, The "FRL" Manual** Memo 408 et 409, Massachussets Institute of Technology, 1977
[TAILLIBERT 86]	P. TAILLIBERT, S. VARENNES, **MI4 : Ou comment développer des systèmes experts en Prolog**, 6ème Journée Internationale sur les Systèmes Experts et leurs applications, AVIGNON (FRANCE), Mai 1986
[VIALATTE 85]	M. VIALATTE, **Description et Application du moteur d'inférence SNARK**, Thèse de Doctorat, Université PARIS VI, Mai 1985

PROCEDURE D'AIDE A LA DECISION MULTI-INFORMATEURS APPLICATION A LA CLASSIFICATION MULTI-CAPTEURS DE CIBLES

Par A. APPRIOU

ONERA
29 avenue de la Division Leclerc
Boîte postale 72
92322 CHATILLON
FRANCE

RESUME

Un processus d'aide à la décision propre à traiter la majorité des problèmes de choix en présence d'informations multiples est élaboré à partir d'une formulation aussi objective et exhaustive que possible de ces problèmes. Le processus décrit, propre à être intégré dans un Système Expert capable de gérer différentes options de façon interactive, est illustré par une application à la classification multi-capteurs de cibles.

Cette application permet de discuter le comportement d'un certain nombre de techniques de fusion de données pour lesquelles une formulation adaptée au problème envisagé est proposée.

L'extension de ces techniques de décision à un processus original d'Allocation de Ressources est ensuite présentée ; elle permet, à partir d'informations préalables sur un certain nombre de situations à analyser simultanément, d'attribuer le traitement d'une de ces situations à chacun des informateurs disponibles, afin de fournir la meilleure évaluation sur l'ensemble des situations. La comparaison des différentes solutions dégagées est abordée par la simulation du scénario de classification déjà utilisé précédemment.

I) INTRODUCTION

L'intelligence Artificielle, et plus particulièrement les Systèmes Experts, semblent devoir apporter une aide appréciable à l'élaboration de choix et de décisions dans le domaine de la défense, que ce soit pour l'analyse de situations, ou pour la planification d'actions à mener. Dans tous les cas, leur utilisation requiert la mise au point de procédures d'agrégation d'informations d'origines diverses permettant l'élaboration des règles de décision, éventuellement stochastiques, qui doivent alimenter leur base de connaissance.

Dans ce contexte, le présent exposé propose un processus d'aide à la décision propre à traiter les applications où les informations à synthétiser permettent l'élaboration de grandeurs caractéristiques de la confiance relative à accorder aux différentes hypothèses possibles, répertoriées à priori. Ces grandeurs peuvent être définies sous forme de données incertaines, qui prennent notamment en compte des informations extérieures relatives à l'environnement, au bon fonctionnement des informateurs utilisés, etc... Le processus décrit peut utiliser différentes lois de fusion élaborées à partir de techniques garantissant transitivité, associativité et relation d'ordre total, telles que l'Inférence Bayésienne, le Maximum d'Entropie, la Théorie de l'Evidence développée par Dempster et Shafer, la Théorie des Ensembles Flous de Zadeh, ou une approche plus directe de classement statistique des éventualités envisageables. Ces différentes possibilités sont comparées dans le cadre de leur application à la classification multi-capteurs de cibles, sur la base de la simulation d'un scénario typique.

Un tel processus de décision est par ailleurs le plus souvent bouclé par une technique d'Allocation de Ressources qui définit en permanence quel moyen d'analyse doit être affecté au traitement des différentes situations à évaluer, afin de garantir le meilleur résultat global, compte tenu de l'information déjà acquise. Plusieurs procédures susceptibles d'assurer cette fonction sont donc proposées et comparées sur la base du même scénario de classification multi-capteur.

II) FORMULATION DU PROBLEME

Conformément à la figure 1, le problème posé suppose que l'on acquiert m informations g_j, $j \in [1, m]$, et que l'on peut établir à partir de chacune d'elles n grandeurs e_{ij}, $i \in [1, n]$, représentatives de l'importance ou de la confiance relative qu'il convient d'attribuer à chacune des n hypothèses H_i envisagées à priori, au vu de la grandeur g_j considérée.

Dans le cadre de cet exposé, les grandeurs e_{ij} élaborées sont supposées indépendantes d'une information g_j à l'autre ; cette hypothèse correspond le plus souvent à un choix judicieux (ou une tranformation) des informations qui garantit leur meilleure complémentarité.

Le problème posé consiste alors à chercher l'hypothèse Hi qui satisfait "au mieux" :

(1) $$\max_i \{e_{ij}\} \ , \forall j \ .$$

Il convient donc de réaliser une méthode d'agrégation des e_{ij} par rapport aux informations j, éventuellement après changement de signe des composantes e_{ij} pour lesquelles le problème initial serait celui d'une minimisation.

Cette méthode doit, de préférence, être capable de traiter les cas où les grandeurs e_{ij} sont incertaines, c'est-à-dire simplement définies par un intervalle de valeurs possibles $[e_{ij}^{min}, e_{ij}^{max}]$, afin de permettre, notamment, la prise en compte de l'aptitude d'une information à émettre un avis sur une hypothèse donnée, et donc de la sureté de cet avis.

Le processus de décision qui préside au choix de l'hypothèse à retenir doit par ailleurs prendre en considération un certain nombre d'informations utiles sur l'environnement et le contexte de l'opération menée, pour être pleinement efficace. Nous supposerons dans la suite que ces informations sont disponibles sous forme de différents indices que nous rassemblons ici de façon synthétique selon trois catégories fonctionnelles :

a) Des indices a_{1j} de qualité de l'information gj, qui représentent l'aptitude de cette information à traiter le problème de décision posé, sa robustesse vis-à-vis de ce problème. Typiquement un tel indice peut figurer la présence de brouillage ou de leurrage pour une mission d'analyse de situation, une éventualité non répertoriée à priori parmi les hypothèses Hi, l'effet d'un mauvais rapport signal/bruit ou d'une atténuation importante au niveau d'un senseur. Un tel indice résulte d'une analyse appropriée des informations elles-mêmes.

b) Des indices a_{2j} de confiance ou d'importance conférée à priori à l'information gj ; ces indices, déterminés à partir de règles à caractère "dictatorial" imposées par l'opérateur ou le Système Expert qui gère le processus de décision présenté dans la suite, ont une fonction analogue à celle des indices a_{1j} ; ils caractérisent simplement la volonté de l'utilisateur d'infléchir la décision finale en prêtant une attention plus grande à certaines informations qu'à d'autres, compte tenu de sa connaissance personnelle du contexte et de l'évolution de celui-ci.

c) Des indices bi de préférence "à priori" des hypothèses mises en jeu ; ces indices traduisent la volonté de l'utilisateur de trancher arbitrairement, si nécessaire, entre plusieurs solutions qui seraient déclarées équivalentes par l'analyse des informations disponibles. Cette possibilité répond à un souci dégagé par K.J. Arrow : celui-ci met en effet en évidence [1 et 2] qu'aucune règle de décision autre que la "dictature" n'est satisfaisante dans l'absolu, et qu'en pratique il convient d'adopter un compromis judicieux entre la rationalité, le caractère décisif et l'égalitarisme.

Dans le cadre du problème traité ici, l'analyse des informations doit donc rester un processus d'aide à la décision dont il convient de pallier les limites légitimes, in fine, par un arbitraire ultime qui traduit la politique générale de l'utilisateur selon la situation traitée (prudence dans les choix, détermination à assurer une finalité vitale...).

Ces trois types d'indice procurent donc des degrés de liberté supplémentaires au processus d'aide à la décision basé sur la fusion des seules évaluations e_{ij} ; ils figurent en particulier les points d'entrée réservés à l'utilisateur ou au Système Expert responsable de sa mise en oeuvre. Ils ne doivent toutefois en aucun cas être surabondants par rapport à la connaissance apportée par les grandeurs e_{ij} .

Notons enfin que ces indices a_{1j} , a_{2j} et b_i , ainsi que les grandeurs e_{ij} , peuvent à priori prendre leurs valeurs indifféremment dans l'un des trois ensembles suivants :

- l'ensemble binaire figurant un avis tout-ou-rien : présent/absent, bon/mauvais, vrai/faux... . Ce sera notamment le cas, pour les e_{ij} , dans une structure décentralisée où chaque informateur prend sa décision indépendamment. Le processus de décision proposé dans la suite fournit alors une simple table de vérité qui établit la procédure de vote à adopter dans les conditions du problème. Toute la difficulté de la décision en temps réel est dans ce cas reportée au niveau local de chaque informateur et doit être résolue par la Théorie des jeux coopératifs (J1 = J2 = ... = JM). Les performances obtenues avec une telle structure sont toutefois nécessairement moins bonnes que celles d'une décision centralisée, lorsque celle-ci est possible.

- l'ensemble des rangs où chaque grandeur prend pour valeur son ordre de priorité. Cette topologie conduit à un traitement non paramétrique des informations. La présence d'ex-aequos permet ici de prendre en compte toutes les partitions possibles en hypothèses (ou informations) équivalentes.

- l'ensemble des réels, la grandeur considérée étant alors une note ou un poids qui permet de prendre en compte une mesure de l'écart séparant deux valeurs successives. Il en est par exemple ainsi lorsque les grandeurs considérées résultent de l'évaluation d'une distance ou d'une probabilité, en particulier dans le cas le plus général où :

(2) $$e_{ij} \equiv P[H_i / g_j]$$

Ces différentes topologies peuvent également co-exister dans un même processus de décision.

III) PROCESSUS DE DECISION

La figure 2 représente le processus de décision séquentiel qui découle logiquement de l'approche introduite précédemment. Il consiste en premier lieu à élaborer les grandeurs caractéristiques e_{ij} et les indices a_{1j}, a_{2j} et b_i. Les hypothèses Hi et les informations gj qui peuvent être définitivement écartées par l'affectation d'un indice minimal, le sont dès que possible afin d'éviter tout calcul inutile, notamment au niveau de l'élaboration des grandeurs e_{ij}. Cette opération nous ramène à N hypothèses Hi possibles ($N \leqslant n$) et M informations disponibles ($M \leqslant m$).

Toutes les grandeurs e_{ij} et les indices a_{1j} et a_{2j} restants doivent ensuite être harmonisés, tant du point de vue topologique qu'en ce qui concerne l'intervalle de variation, afin de pouvoir être confrontés. La transformation mise en oeuvre doit toutefois reproduire sans distorsion l'étalement relatif des valeurs susceptibles d'être observées, cet étalement étant ici réputé parfaitement représentatif de la caractéristique traitée (éventualité, volonté, ...) ; elle doit donc conserver toute relation d'ordre sur l'ensemble des valeurs concernées, ainsi que toute relation d'ordre entre les accroissements sur les éléments de cet ensemble ; elle est donc linéaire, soit :

$$(3) \qquad \left.\begin{matrix}\mathbb{R}\\ \mathbb{N}\\ \mathbb{B}\end{matrix}\right\} \xrightarrow{\mathcal{L}} [0,1]$$

l'intervalle [0, 1] étant arbitrairement retenu pour sa compatibilité immédiate avec certaines notions utilisées ultérieurement (fonctions d'appartenance, fonctions d'utilité, probabilités, ...). Typiquement, elle conduit à adopter :

$$(4) \qquad \varepsilon_{ij} = \frac{e_{ij} - e_{j\min}}{e_{j\max} - e_{j\min}} \qquad \text{et (5)} \qquad \alpha_{ij} = \frac{a_{ij} - a_{i\min}}{a_{i\max} - a_{i\min}}$$

où

$$(6)\ e_{j\min} = \min_{i\in[1,n]}\{e_{ij}\}\ ,\ (7)\ e_{j\max} = \max_{i\in[1,n]}\{e_{ij}\}\ ,\ (8)\ a_{i\min} = \min_{j\in[1,m]}\{a_{ij}\}\ ,\ (9)\ a_{i\max} = \max_{j\in[1,m]}\{a_{ij}\}$$

résultent de la connaissance des caractéristiques et des limites physiques des grandeurs traitées, de contraintes opérationnelles (saturations, ...), ou d'une détermination empirique directe à partir de valeurs observées. Dans l'alternative où ces extréma ne sauraient être évalués, l'emploi de transformations non linéaires (exponentielles, ...) doit être manié avec discernement compte tenu des effets qu'elles induisent dans le processus. Notons que l'emploi des rangs affranchit de ce dilemme dans tous les cas.

Il convient alors de réaliser en premier lieu l'agrégation des indices α_{1j} et α_{2j} puisqu'ils remplissent la même fonction, par une transformation

$$(10) \qquad \alpha_j = f(\alpha_{1j}, \alpha_{2j})$$

procurant une priorité globale pour chaque information disponible. Il est ainsi possible d'envisager la fusion inter-informations des grandeurs ε_{ij}, en prenant en compte pour chacune d'elle sa priorité α_j :

$$(11) \qquad C_i = g(\alpha_j, \varepsilon_{ij}^{\min}, \varepsilon_{ij}^{\max})$$

La recherche du, ou des meilleurs coûts Ci ainsi obtenus procure alors la, ou les hypothèses retenues par l'analyse des informations disponibles.

L'hypothèse "optimale" Hi* est finalement choisie parmi les No hypothèses dégagées par ce processus (No < N), dans la mesure où celà s'avère nécessaire, grâce au classement b_i fourni arbitrairement par les règles "dictatoriales" conformément à la philosophie dégagée au paragraphe II.

Le processus détaillé ici peut également être itéré de façon interactive avec le Système Expert qui le gère, et qui peut, en fonction des hypothèses dégagées, demander une autre analyse avec des lois de fusions, des priorités ou des topologies différentes.

Un module d'allocation de ressources, lui aussi géré par le Système Expert, peut enfin définir les informations qu'il convient de chercher à recueillir au vu des évaluations antérieures. Une recherche et une discussion de techniques applicables à ce module dans un contexte plus global où un nombre limité de moyens d'analyse doivent permettre la meilleure évaluation simultanée d'un certain nombre de situations, sont proposées dans le cadre de cet exposé.

Dans l'immédiat, nous allons chercher à préciser la transformation analytique (11) qui assure la fusion des évaluations disponibles, en applicant différentes théories d'inférence au problème ainsi défini. Notons que la loi de fusion (10) peut être considérée comme une application triviale des formulations qui vont êre dégagées pour (11) : elle ne fera donc pas l'objet d'une investigation particulière.

IV) LOIS DE FUSION

Le problème est ici d'élaborer une fonction de coût Ci dont l'optimisation permette de déterminer l'hypothèse Hi qui maximise le vecteur $E_i = (\varepsilon_{i1}, \varepsilon_{i2}, \dots, \varepsilon_{iM})$ (12), à composantes éventuellement incertaines, conformément au processus présenté au paragraphe III. La loi de fusion cherchée doit donc fournir une relation d'ordre total transitive sur les hypothèses en compétition et être de surcroît associative par rapport aux informations disponibles (résultat indépendant de l'ordre de fusion des informations), ce qui exclut d'emblée un certain nombre de techniques [3, 7, 8 en particulier]. Dans cette optique plusieurs approches sont à priori envisagées.

IV.1) INFERENCE BAYESIENNE

Le critère optimisé est ici :

(13) $C_i = P[H_i / g_1, g_2, \cdots, g_M]$

La règle de Bayes fournit, pour des informations gj indépendantes :

(14) $C_i = P(H_i) . \prod_{j=1}^{M} P[g_j / H_i] \Big/ \sum_{i=1}^{N} P(H_i) . \prod_{j=1}^{M} P[g_j / H_i]$

Dans le cadre général où :

(15) $\varepsilon_{ij} \equiv P[g_j / H_i]$

Ci s'exprime, pour des hypothèses équiprobables à priori :

(16) $C_i = \prod_{j=1}^{M} \varepsilon_{ij} \Big/ \sum_{i=1}^{N} \left(\prod_{j=1}^{M} \varepsilon_{ij} \right)$

Le dénominateur de cette expression étant indépendant de i, Ci peut s'écrire, si l'on interprète *parallèlement* α_j comme la fonction d'utilité associée à l'entropie de chaque terme :

(17) $C_i = \prod_{j=1}^{M} (\varepsilon_{ij})^{\alpha_j}$

Il est toutefois à noter que cette méthode est impropre à traiter des données incertaines, et qu'elle exige la *connaissance exhaustive de tous les* ε_{ij} au moment de la décision.

IV.2) MAXIMUM D'ENTROPIE

Différentes approches peuvent être envisagées dans ce domaine [11]. La méthode la plus fréquemment adoptée utilise une mesure de la quantité moyenne d'information disponible *pour chaque* hypothèse. Les informations gj étant indépendantes, elle conduit, pour une même définition (15) des ε_{ij} *qu'au paragraphe précédent, à retenir :*

(18) $C_i = \sum_{j=1}^{M} \alpha_j \, \varepsilon_{ij} \operatorname{Log} \varepsilon_{ij}$

où α_j est interprété comme la fonction d'utilité associée à chaque informateur. Cette méthode ne se distingue donc de l'Inférence Bayésienne que par une pondération supplémentaire en ε_{ij} ; ce léger surcroît de complexité ne peut donc se justifier que par des performances *suffisamment accrues*, les remarques faites alors sur les conditions d'emploi restant valables ici.

IV.3) CLASSEMENT STATISTIQUE DIRECT

Déterminer l'hypothèse Hi qui maximise le vecteur Ei = $(\varepsilon_{i1}, \cdots, \varepsilon_{iM})$ suppose la définition d'une relation d'ordre qui permette de comparer les vecteurs Ei entre eux. Or, si l'on ne veut pas particulariser le problème par l'emploi d'une norme spécifique, *la seule relation d'ordre qu'il est raisonnable* d'adopter admet qu'un vecteur est supérieur à un autre si toutes leurs composantes respectives *suivent ce classement, soit plus précisément :*

(19) $E_i > E_\ell \iff \varepsilon_{ij} > \varepsilon_{\ell j} \; , \; \forall j \in [1, M]$

Cette relation n'est toutefois pas une relation d'ordre total dès lors que $M > 1$, et la maximisation de Ei peut s'exprimer de deux façon différentes, qui ne sont pas équivalentes : soit chercher à avoir un minimum de vecteurs supérieurs au vecteur à retenir, soit chercher à avoir un maximum de vecteurs inférieurs. En termes probabilistes, ceci s'exprime par :

(20) $\min_{i \in [1,N]} \left\{ P[E_k , k \in [1,N], E_k > E_i] \right\}$

et

(21) $\max_{i \in [1,N]} \left\{ P[E_k , k \in [1,N], E_k < E_i] \right\}$

Si u_{ij} désigne la valeur qu'est susceptible de prendre la *grandeur* ε_{ij}, *l'indépendance* des composantes ε_{ij} d'un même vecteur Ei conduit à exprimer (20) et (21) respectivement par :

(22) $\min_{i \in [1,N]} \left\{ C_{i1} \right\}$

et

(23) $\max_{i \in [1,N]} \left\{ C_{i2} \right\}$

avec

(24) $C_{i1} = \prod_{j=1}^{M} \left(\sum_{\ell=1}^{N} P[H_\ell] \int_{\varepsilon_{ij}}^{1} P[u_{ij} / H_\ell] \, du_{ij} \right)$

et

(25) $C_{i2} = \prod_{j=1}^{M} \left(\sum_{\ell=1}^{N} P[H_\ell] \int_{0}^{\varepsilon_{ij}} P[u_{ij} / H_\ell] \, du_{ij} \right)$

Par extension, dans le cas de grandeurs ε_{ij} incertaines, les *conditions* (20) et (21) *doivent* être remplies pour tous les vecteurs Ei dont les composantes ε_{ij} sont respectivement comprises dans les intervalles $[\varepsilon_{ij}^{min}, \varepsilon_{ij}^{max}]$; elles doivent donc être définies pour le vecteur Ei le plus contraignant dans chaque cas, à savoir $\varepsilon_{ij} = \varepsilon_{ij}^{min}$ pour (20) et $\varepsilon_{ij} = \varepsilon_{ij}^{max}$ pour (21). Les critères C_{i1} et C_{i2} à optimiser selon (22) et (23) deviennent dans ces conditions :

(26) $C_{i1} = \prod_{j=1}^{M} \left(\sum_{\ell=1}^{N} P[H_\ell] \int_{\varepsilon_{ij}^{min}}^{1} P[u_{ij} / H_\ell] \, du_{ij} \right)$

et

(27) $C_{i2} = \prod_{j=1}^{M} \left(\sum_{\ell=1}^{N} P[H_\ell] \int_{0}^{\varepsilon_{ij}^{max}} P[u_{ij} / H_\ell] \, du_{ij} \right)$

Ces deux critères encadrent en fait l'optimum d'une "qualité minimale garantie" (C_{i2}) et d'une "performance maximale possible" (C_{i1}). Ils regroupent donc à eux deux le potentiel nécessaire et suffisant pour définir un optimum au sens de la relation d'ordre (19). Toutefois, pour répondre au problème posé, il convient encore de les fusionner en un critère unique Ci à maximiser. Afin d'harmoniser les effets sur Ci des dynamiques de chacun des critères, il est souhaitable de lui imposer la conservation des variations relatives des deux critères C_{i1} et C_{i2}, soit, compte tenu de (22) et (23):

(28) $$\frac{dC_i}{C_i} = -\frac{dC_{i1}}{C_{i1}}$$

et

(29) $$\frac{dC_i}{C_i} = \frac{dC_{i2}}{C_{i2}}$$

L'intégration de (28) et (29) fournit ainsi pour notre problème de maximisation :

(30) $$C_i = C_{i2} / C_{i1}$$

Toutefois les expressions (26) et (27) sont souvent difficiles à établir en pratique, par méconnaissance des densités $P[u_{ij}/H_\ell]$. Leur application au cas particulier où la densité $\sum_\ell P[H_\ell] \cdot P[u_{ij}/H_\ell]$ est uniforme sur [0, 1] permet cependant de satisfaire au mieux le cas où les informations gj sont bien adaptées au problème de décision posé, produisant une dispersion maximale des ε_{ij} pour l'ensemble des hypothèses Hi, équiprobables à priori. Or ce cas est celui qui nous intéresse puisque l'aptitude de chaque informateurs à discriminer les différentes hypothèses est "filtré" par ailleurs (α_j ; cf II et III). Notons que cette simplification est plus spécialement justifiée avec l'emploi des rangs pour ε_{ij} ; elle conduit dans tous les cas à retenir :

(31) $$C_i = \prod_{j=1}^{n} \frac{\varepsilon_{ij}^{max}}{(1-\varepsilon_{ij}^{min})}$$

Par ailleurs, une diminution du facteur de qualité α_j doit se traduire par suspicion croissante sur l'évaluation ε_{ij} fournie, et donc par une augmentation de l'incertitude sur cette grandeur. Cette variation doit en outre refléter sans distorsion l'étalement relatif des valeurs α_j, réputé parfaitement représentatif du doute à introduire ; elle est donc linéaire :

(32) $$\varepsilon_{ij}^{min} \longrightarrow \alpha_j \; \varepsilon_{ij}^{min}$$

(33) $$\varepsilon_{ij}^{max} \longrightarrow 1 - \alpha_j(1 - \varepsilon_{ij}^{max})$$

Notons enfin que cette technique permet, contrairement aux précédentes, de traiter les applications où à l'instant de décision toutes les évaluations ε_{ij} ne sont pas encore connues, en affectant aux inconnues une incertitude maximale : $\varepsilon_{ij}^{min} = 0$ et $\varepsilon_{ij}^{max} = 1$.

IV.4) THEORIE DE L'EVIDENCE

Cette technique, dont les fondements théoriques ont été posés par Dempster et Shafer [12 à 14] et dont l'utilité a été plus récemment démontrée sur des applications pratiques [15 à 18], est en fait une généralisation de l'Inférence Bayésienne au traitement des données incertaines. Avant d'en produire l'application au problème posé ici, quelques notions de base sont brièvement rappelées.

IV.4.1) PRESENTATION SUCCINCTE

La méthode suppose N hypothèses exhaustives Hi s'excluant mutuellement, qui constituent un ensemble E, appelé "cadre de discernement". Une masse m(Hi) est attribuée à chaque hypothèse pour quantifier sa propre probabilité, et une masse m(E) est assignée à l'ensemble E pour matérialiser l'incertitude que l'on peut avoir à désigner l'une ou l'autre quelconque des hypothèses Hi de E. Ces masses doivent satisfaire la condition :

(34) $$\sum_{i=1}^{N} m(H_i) + m(E) = 1$$

De plus, pour tout ensemble $A \subset E$, nous avons :

(35) $$m(A) = \sum_{H_i \subset A} m(H_i)$$

D'une façon plus générale, pour tout découpage de E en sous-ensembles exclusifs Aj d'hypothèses, deux notions caractérisent un sous-ensemble quelconque A de E :

- Le support de A, qui représente la probabilité avec laquelle on peut affirmer A :

(36) $$S(A) = \sum_{A_j \subset A} m(A_j)$$

- La plausibilité de A, qui représente la probabilité de son éventualité, c'est-à-dire la probabilité avec laquelle on ne peut affirmer le contraire $\bar{A}$ de A ($A \cup \bar{A} = E$) :

(37) $$P(A) = \sum_{A_j \not\subset \bar{A}} m(A_j) = 1 - S(\bar{A})$$

Sur ces bases, la théorie de l'évidence fournit des règles d'inférence qui permettent de combiner des informations provenant de sources indépendantes. Leur principe de base consiste, pour deux sources produisant respectivement des jeux de masses $\{m_1(A_i)\}$ (dont $m_1(E)$) et $\{m_2(B_j)\}$ (dont $m_2(E)$) relativement à deux découpages quelconques de E en sous-ensembles exclusifs, à déduire un jeu de masses tel que :

(38) $$m(C_\ell) = \sum_{A_i \cap B_j = C_\ell} m_1(A_i)\, m_2(B_j) / (1-k)$$

où k est "l'inconsistance" de la fusion :

(39) $$k = \sum_{A_i \cap B_j = \emptyset} m_1(A_i)\, m_2(B_j)$$

Cette méthode autorise en particulier la fusion de M sources ($M > 2$), indépendamment de leur ordre de fusion, ce qui satisfait le besoin d'associativité pour notre problème. Elle présente en outre l'avantage sur les méthodes précédentes de permettre la fusion de découpages Ai et Bj différents dans le cadre de discernement E.

IV.4.2) APPLICATION AU PROBLEME POSE

Pour ramener le problème étudié au formalisme de la Théorie de l'Evidence, les grandeurs ε_{ij} doivent être assimilées à une mesure de vraisemblance des différentes hypothèses Hi, pour lesquelles un support et une plausibilité sont alors associés à chaque information gj :

(40) $S_{ij}(H_i) = \varepsilon_{ij}^{min} = m_{ij}(H_i)$

(41) $P_{ij}(H_i) = \varepsilon_{ij}^{max}$

Il convient dans ces conditions de réaliser l'agrégation des S_{ij} et P_{ij} de façon à obtenir un support S(Hi) et une plausibilité P(Hi) globaux pour chaque hypothèse Hi, dont il faut ensuite assurer la synthèse sous la forme d'un critère unique Ci à maximiser.

Bien que la Théorie de l'Evidence reste muette au niveau de cette dernière synthèse, il semble légitime dans le cadre du problème traité de chercher l'hypothèse la plus plausible pour laquelle, corrélativement, l'éventualité d'une quelconque autre hypothèse est la moins probable. Ceci revient donc, en vertue des notions rappelées au paragraphe précédent, à optimiser conjointement :

(42) $\min\limits_{i \in [1,N]} \{C_{i1}\}$

et

(43) $\max\limits_{i \in [1,N]} \{C_{i2}\}$

avec

(44) $C_{i1} = P(\overline{H_i}) = 1 - S(H_i)$

et

(45) $C_{i2} = P(H_i)$

Si de surcroit on veut établir un critère global Ci qui conserve la dynamique de chacun des critères C_{i1} et C_{i2}, on est amené, ici encore, à imposer l'identité des variations relatives :

(46) $dC_i / C_i = - dC_{i1} / C_{i1}$

et

(47) $dC_i / C_i = dC_{i2} / C_{i2}$

qui conduit par intégration à :

(48) $C_i = C_{i2} / C_{i1} = P(H_i) / (1 - S(H_i))$

La détermination de S(Hi) et P(Hi) par les règles d'inférence de la Théorie de l'Evidence est explicitée en annexe.

Elle conduit à :

(49) $S(H_i) = 1 - \left\{ \prod\limits_{j=1}^{M} [1 - S_{ij}(H_i)] \right\} / k_i$ et (50) $P(H_i) = \left\{ \prod\limits_{j=1}^{M} P_{ij}(H_i) \right\} / k_i$

avec

(51) $k_i = \prod\limits_{j=1}^{M} P_{ij}(H_i) + \prod\limits_{j=1}^{M} [1 - S(H_i)] - \prod\limits_{j=1}^{M} [P_{ij}(H_i) - S_{ij}(H_i)]$

L'expression (48) du critère Ci affranchit en fait du calcul de ki, et les définitions (40) et (41) de S_{ij} et P_{ij} procurent finalement :

(52) $$C_i = \prod_{j=1}^{M} \frac{\varepsilon_{ij}^{max}}{(1 - \varepsilon_{ij}^{min})}$$

Il est à remarquer que cette loi de fusion est la même que celle obtenue par le classement statistique direct. La prise en compte de α_j doit par ailleurs se faire de la même façon par (32) et (33), et cela pour les mêmes raisons. Les conditions opérationnelles d'emploi sont enfin identiques.

IV.5) THEORIE DES ENSEMBLES FLOUS

IV.5.1) GENERALITES

Afin de permettre la prise en comtpe de l'imprécis dans un raisonnement, quelle qu'en soit la nature, L.A. Zadeh propose de définir sur un ensemble E d'éléments X des sous-ensembles flous A à l'aide d'une fonction d'appartenance $\mu_A(X)$ [19 à 22] :

$$\mu_A \begin{cases} X \longrightarrow \mu_A(X) \\ E \longrightarrow [0,1] \end{cases}$$

Un élément X est d'autant plus considéré comme un élément de l'ensemble A que sa fonction d'appartenance $\mu_A(X)$ est grande. Les valeurs extrêmes 0 et 1 de cette fonction correspondent à la notion binaire d'appartenance de la théorie "classique" des ensembles. La théorie des ensembles flous permet d'introduire, par rapport à celle-ci, le fait qu'un élément appartient "plus ou moins" à un ensemble A, qu'un évènement est plus ou moins sûr... . Un ensemble flou est donc un ensemble de couples $(X, \mu A(X)) \in E \times [0, 1]$. Notons toutefois qu'une fonction d'appartenance n'est en général pas définie de façon unique, son interprétation dépendant du contexte.

Les notions classiquement utilisées pour les ensembles se traduisent ici par des relations sur les fonctions d'appartenance ; *parmi les principales, rappelons :*

- l'inclusion : $A \subset B \Longleftrightarrow \forall X \in E \ , \ \mu_A(X) \leqslant \mu_B(X)$
- l'égalité : $A = B \Longleftrightarrow \forall X \in E \ , \ \mu_A(X) = \mu_B(X)$
- la complémentarité : $A \cap \bar{A} = \emptyset$ et $A \cup \bar{A} = E \Longleftrightarrow \forall X \in E \ , \ \mu_{\bar{A}}(X) = 1 - \mu_A(X)$
- l'union : $\forall X \in E \ , \ \mu_{A \cup B}(X) = \max[\mu_A(X), \mu_B(X)]$
- l'intersection : $\forall X \in E \ , \ \mu_{A \cap B}(X) = \min[\mu_A(X), \mu_B(X)]$

La combinaison convexe C de A et B est par ailleurs définie par :

$$\mu_C(X) = \gamma \, \mu_A(X) + (1-\gamma)\mu_B(X) \quad , \quad \gamma \in [0,1]$$

Cette théorie, applicable à différents domaines, apporte deux types de solution aux problèmes de décision :

1) Pour un ensemble X d'hypothèses Hi, des sous-ensembles X_{G_j} de finalités Gj à satisfaire parmi ces hypothèses et des sous-ensembles X_{C_ℓ} d'hypothèses résultant de contraintes Cl, l'ensemble à retenir est :

(53) $$X_D = \left(\bigcap_j X_{G_j}\right) \cap \left(\bigcap_\ell X_{C_\ell}\right)$$

et la meilleure *décision Hi à prendre est donnée par :*

(54) $$\max_i \left\{ \mu_{X_D}(H_i) \right\}$$ où, donc : $\mu_{X_D}(H_i) = \min_{j,\ell} \left\{ \mu_{X_{G_j}}(H_i), \mu_{X_{C_\ell}}(H_i) \right\}$

Nous désignerons dans la suite cette technique par l'intitulé "Décision par Finalité Floue" (DFF).

2) Dans le même contexte que ci-dessus, une décision pondérée conduit à retenir une combinaison convexe des ensembles X_{G_j} et X_{C_ℓ} ; la décision Hi est alors désignée par :

(55) $$\max_i \left\{ \mu_{X_D}(H_i) \right\}$$ où : $\mu_{X_D}(H_i) = \sum_j \beta_j \, \mu_{X_{G_j}}(H_i) + \sum_\ell \gamma_\ell \, \mu_{X_{C_\ell}}(H_i)$
avec : $\sum_j \beta_j + \sum_\ell \gamma_\ell = 1$

Cette *technique sera labellée, dans la suite :* " Décision Convexe Floue" (DCF).

IV.5.2) <u>APPLICATION AU PROBLEME POSE</u>

Dans notre cas, il convient de satisfaire un certain nombre de finalités que sont les avis donnés par les différents informateurs. Un ensemble X_{G_j} est donc pour nous l'ensemble des hypothèses retenues par l'informateur gj comme étant les plus satisfaisantes. De façon immédiate, la fonction d'appartenance d'une hypothèse Hi à l'ensemble X_{G_j} est directement donnée par *la grandeur* ε_{ij} *qui remplit* toutes les conditions requises pour cela.

En l'absence de contraintes, la DFF conduit à retenir l'intersection des avis émis par les *différents informateurs,* et à désigner l'hypothèse Hi qui satisfait, d'après (54) :

(56) $$\max_i \left\{ C_i \right\} \quad , \quad C_i = \min_j \left\{ \varepsilon_{ij} \right\}$$

Dans le cas de grandeurs ε_{ij} incertaines, les valeurs comprises entre ε_{ij}^{min} et ε_{ij}^{max} doivent être considérées comme autant d'avis possibles à satisfaire ; l'expression (56) devient alors :

(57) $$\max_i \left\{ C_i \right\} \quad , \quad C_i = \min_j \left\{ \varepsilon_{ij}^{min} \right\}$$

Parallèlement, la DCF *conduit dans* le *même* contexte, et pour des avis équipollents, à retenir l'hypothèse Hi désignée par (55) :

(58) $$\max_i \left\{ C_i \right\} \quad , \quad C_i = \frac{1}{2M} \sum_{j=1}^{M} \left(\varepsilon_{ij}^{min} + \varepsilon_{ij}^{max} \right)$$

L'indice de qualité α_j est ici aussi supposé intégré dans l'intervalle d'incertitude par (32) et (33). Notons qu'en ce qui concerne (58) une autre attitude pourrait consister à ne pas intégrer α_j dans l'intervalle d'incertitude et à l'utiliser comme coëfficient de la combinaison *convexe* ; les deux attitudes conduisent toutefois rigoureusement au même résultat.

V) <u>APPLICATION A LA CLASSIFICATION MULTI-CAPTEURS DE CIBLES</u>

V.1) <u>PRESENTATION GENERALE</u>

Conformément à la figure 3, l'élaboration des grandeurs e_{ij}, qui représentent les poids affectés à chacune des classes possibles i par chacun des capteurs j, fait l'objet d'un pré-traitement propre à chaque capteur, décomposable en deux étapes : la première consiste à extraire, à partir des signaux ou des images relevés par le capteur, les grandeurs caractéristiques de l'objet observé qui sont invariantes aux conditions de mesure. Les valeurs obtenues sont alors *comparées aux grandeurs* de même nature apprises au préalable sur chaque type de cible possible, à l'aide de discriminateurs propres à évaluer *leur degré* de similitude *avec chacun de ces apprentissages,* la grandeur e_{ij} cherchée étant directement représentative de cette similitude. Typiquement, ce pré-traitement peut être réalisé par *filtrage inverse ou par filtrage adapté* de signaux invariants, où chaque filtre est adapté à une classe de cible ; dans le cas d'images, ce peut être la comparaison par une distance statistique de moments, ou la corrélation de contours, de squelettes, ou de textures pour chaque classe possible.

Les grandeurs e_{ij} obtenues sont alors fusionnées selon le processus présenté précédemment, qui permet la prise en compte du contexte, de l'environnement, et de toute connaissance *extérieure ou* préalable pour conduire au choix de la classe la plus probable.

La plupart des méthodes abordées permettant en outre la prise en compte de grandeurs e_{ij} incertaines, il est ici proposé d'utiliser un éventuel apprentissage de la matrice de confusion de chaque capteur pour déterminer l'incertitude liée à chaque évaluation e_{ij}. Le terme général P_{ik}^{j} d'une telle matrice, associée au capteur j, est en effet représentatif de la probabilité qu'à ce capteur de reconnaître la cible *k* lorsque la *cible i lui est présentée* :

(59) $$P_{ik}^{j} = P\left[e_{kj} = \max_{\ell\in[1,N]}\{e_{\ell j}\} \,/\, \text{cible } i\right]$$

Son terme *diagonal* P_{ii}^{j} *figure donc directement le degré d'aptitude de la grandeur* e_{ij} *à bien* reconnaître la cible i ; toute diminution de sa part doit par conséquent se traduire par une suspicion *croissante sur l'évalutation* e_{ij} *fournie*, c'est-à-dire par une augmentation de l'incertitude sur cette grandeur. Cette variation doit en outre refléter sans distorsion l'étalement relatif des P_{ii}^{j}, réputé parfaitement représentatif du doute à introduire ; elle est donc linéaire :

(60) $$\begin{cases} e_{ij}^{min} = P_{ii}^{j}(e_{ij} - e_{jmin}) + e_{jmin} \\ e_{ij}^{max} = e_{jmax} - P_{ii}^{j}(e_{jmax} - e_{ij}) \end{cases}$$

Notons qu'en toute rigueur cette façon de procéder est adaptée à un choix correct des discriminateurs, qui garantit aux termes diagonaux de la matrice de confusion d'être maximaux pour l'hypothèse testée ; si tel n'était pas le cas, il conviendrait d'affecter à l'évaluation de l'hypothèse Hi le résultat de l'évalutation e_{kj} qui présente *la plus forte probabilité* P_{ik}^{j} de reconnaissance de cette hypothèse Hi, et corrélativement d'utiliser la probabilité P_{ik}^{j} pour l'*élaboration de son incertitude*. Il faut toutefois être conscient qu'alors peuvent naître des conflits entre hypothèses, qui ne pourront être résolus que par un apport extérieur d'information ou de consignes "dictatoriales".

V.2) <u>EXEMPLE DE MISE EN OEUVRE. COMPARAISON DES DIFFERENTES SOLUTIONS</u>

A des fins de pragmatisme, les différentes solutions dégagées au IV sont appliquées dans le cas d'un *scénario* simple de classification de 3 cibles à l'aide de 2 capteurs, imaginé pour l'occasion. Les matrices de confusion de chacun des capteurs sont données en figure 4. Il est à noter l'aptitude du capteur 1 à réconnaitre la cible 1, mais son incompétence en ce qui concerne les cibles 2 et 3 ; inversement le capteur 2 montre une bonne aptitude *pour la cible* 3, *mais une certaine médiocrité pour* les cibles 1 et 2.

La figure 5 présente une séquence d'évaluations ε_{ij} obtenue à l'aide des deux capteurs, successivement sur les trois cibles. Ces valeurs ont été sélectionnées en accord avec les matrices de confusion présentées plus haut : on peut en particulier vérifier qu'en *fonctionnement mono-capteur le* capteur 1 ne reconnaît que la cible 1 et le capteur 2 ne reconnaît que la cible 3, les bonnes reconnaissances étant entourées *en trait plein et les fausses détections en trait discontinu.*

La figure 6 présente les résultats obtenus pour les trois mêmes essais en fonctionnement multi-capteurs, à l'aide de l'Inférence Bayésienne (IB), du Maximum d'Entropie (ME), et, à titre de référence, de la Théorie de l'Evidence (DS) appliquée aux données certaines ($\varepsilon_{ij}^{min} = \varepsilon_{ij}^{max} = \varepsilon_{ij}$). Les trois méthodes permettent de conserver, dans ce cas, toutes les *compétences individuelles en* reconnaissant les cibles 1 et 3.

En revanche, elles restent toutes impuissantes à reconnaître la cible 2 par déduction. Si cependant on utilise comme critère secondaire de comparaison des trois méthodes l'écart relatif entre les deux meilleures valeurs du coût Ci pour les essais ayant conduit à une bonne reconnaissance, la palme de la meilleure robustesse revient à la Théorie de l'Evidence.

La figure 7 montre les résultats obtenus dans les mêmes conditions à l'aide de la Théorie de l'Evidence (DS), de la Décision par Finalité Floue (DFF) et de la Décision Convexe Floue (DCF), lorsqu'on utilise des données incertaines établies à l'aide de (60) à partir des ε_{ij} du tableau de la figure 5 et des matrices de confusion de la figure 4. La mise en oeuvre de données incertaines permet maintenant de reconnaître les trois cibles. A noter également *sur l'exemple traité le mauvais* comportement de la DFF, et toujours la plus grande robustesse de la Théorie de l'Evidence, au sens du *critère secondaire défini plus haut.*

Une étude statistique du comportement de ces méthodes en fonction de la qualité de chacun des capteurs est illustrée par les figures 8 et 9 dans le cas de 2 cibles et 2 capteurs. Chaque résultat fait l'objet de 1000 tirages aléatoires des e_{ij} selon une densité uniforme dont les bornes assurent les différents taux de *reconnaissance* P_{ii}^{j} *rappelés en regard* pour chaque situation testée. La figure 8 présente les taux de reconnaissance moyens obtenus par les trois méthodes retenues plus haut pour le traitement des données *certaines*, et la *figure* 9 *montre* les mêmes estimations pour les trois méthodes de traitement des données incertaines. Dans chaque cas les performances correspondantes obtenues à l'aide du meilleur capteur seul et du moins bon capteur seul sont rappelées à titre de référence : un fonctionnement multi-capteurs étant en général choisi *pour la bonne complémentarité des différents* capteurs, il est à priori souhaitable qu'il garantisse un fonctionnement au moins aussi bon que le *meilleur des capteurs pris individuellement.*

L'attention du lecteur doit cependant être attirée sur le fait que les cas de mauvais fonctionnement choisis ici sont particulièrement draconiens (P_{ii}^{j} = 0,25) par rapport à un mauvais fonctionnement classiquement observé en reconnaissance, qui conduit plutôt à l'indifférence (P_{ii}^{j} = 0,5). Ils présentent toutefois la particularité de bien mettre en évidence l'avantage et la robustesse de la technique de fusion élaborée à partir de la Théoriede l'Evidence, ainsi que le bien fondé de cette façon de prendre en compte un apprentissage préalable des matrices de confusion des différents capteurs sous forme de données incertaines. Le résultat obtenu par cette technique est par ailleurs tout-à-fait satisfaisant en regard des performances du meilleur des capteurs, pris individuellement. Il est en revanche à déplorer un mauvais comportement de la DFF, bien que ses bienfaits aient pu être appréciés par ailleurs dans des applications très précises, et qui est vraisemblablement dû à la prise en compte quelque peu sommaire et expéditive de l'information disponible qu'elle est amenée à opérer dans ce contexte.

VI) PROCEDURE D'ALLOCATION DE RESSOURCES

Le problème abordé ici est la recherche de la meilleure attribution de M moyens d'analyse j au traitement de L situations l à analyser simultanément, à partir d'une connaissance préalable de chaque situation, afin d'être en mesure de fournir la meilleure évaluation sur l'ensemble des situations. La connaissance préalable utilisée peut provenir d'informations extérieures au système, ou d'une première analyse inopinée par des informations non nécessairement appropriées.

Les données utiles sont de trois types, définis de la façon suivante :

- U_i^{ℓ} = l'utilité "à priori" de déclarer l'hypothèse Hi pour la situation l ($0 \leqslant U_i^{\ell} \leqslant 1$) ; il est par exemple plus urgent de reconnaître un hostile proche, qu'une cible lointaine ou peu dangereuse.

- P_i^{ℓ} = le poids, éventuellement incertain, affecté présentement à l'éventualité Hi pour la situation l ($0 \leqslant P_i^{\ell} \leqslant 1$).

- P_{ik}^{j} = la probabilité, pour l'informateur j, de déclarer l'hypothèse Hk en présence de l'hypothèse Hi ; c'est donc le degré d'aptitude de la déclaration Hk à discriminer l'hypothèse Hi, pour le capteur j. Typiquement, pour un problème de reconnaissance, P_{ik}^{j} est le terme général de la matrice de confusion, déjà présentée au V.

Deux approches vont maintenant être proposées pour résoudre ce problème : une approche "classique" par Inférence Bayésienne, qui nous servira de référence, puis une technique utilisant la fusion de données incertaines, issue de l'étude qui précède. Ces deux méthodes seront comparées sur la base d'un scénario simple de reconnaissance de cibles.

VI.1) SOLUTION BAYESIENNE

Dans cette approche, toutes les données préalables sont nécessairement des données "certaines" et les P_i^{ℓ} sont supposés normalisés :

(61) $\sum_i P_i^{\ell} = 1$

Il est alors possible de définir la Fonction d'Utilité présente de la situation l :

(62) $U_o^{\ell} = \max_i \{ P_i^{\ell} \cdot U_i^{\ell} \}$

puis sa Fonction d'Utilité moyenne, pour toutes les déclarations Hk possibles, si on lui affectait l'informateur j :

(63) $U_j^{\ell} = \sum_k P[\text{déclarer } H_k] \times \max_i \{ P[H_i / H_k \text{ déclarée}] \cdot U_i^{\ell} \}$

qui devient, grace à la règle de Bayes :

(64) $U_j^{\ell} = \sum_k \max_i \{ P_{ik}^{j} \times P_i^{\ell} \times U_i^{\ell} \}$

On peut ainsi déterminer la Fonction d'Utilité Marginale liée à l'association de l'informateur j et de la situation l : $\Delta U_j^{\ell} = U_j^{\ell} - U_o^{\ell}$

Il convient alors de comparer les H combinaisons h possibles entre les M informateurs j et les L situations l = h(j) sur la base de leur meilleure Fonction d'Utilité Marginale moyenne, afin de désigner la meilleure d'entre elles :

(65) $\max_h \{ C_h \}$ avec : $C_h = \sum_j \Delta U_j^{h(j)}$

Cette méthode présente toutefois la particularité de ne laisser aucune place à l'incertitude (notamment sur les P_i^{ℓ} et U_i^{ℓ}), de supposer implicitement une Inférence Bayésienne, et de n'optimiser qu'une Fonction d'Utilité moyenne, ce qui ne sont pas les gages des meilleures performances, comme nous avons déjà pu le constater. La méthode qui suit propose un remède à ces lacunes.

VI.2) SOLUTION PROPOSEE

Pour répondre aux différents soucis dégagés par la méthode précédente, nous pouvons définir une Fonction d'Utilité présente de la situation l de façon incertaine :

$$(66)\qquad \begin{cases} U_o^{\ell\,min} = \max\limits_i \{ P_i^{\ell\,min} \cdot U_i^{\ell\,min} \} \\ U_o^{\ell\,max} = \max\limits_i \{ P_i^{\ell\,max} \cdot U_i^{\ell\,max} \} \end{cases}$$

De même, la Fonction d'Utilité "à postériori" d'associer l'informateur j à la situation l peut être bornée par :

$$(67)\qquad \begin{cases} U_j^{\ell\,min} = \min\limits_k \{ \max\limits_i \{ P_{ik}^{j\ell} \cdot U_i^{\ell\,min} \} \} \\ U_j^{\ell\,max} = \max\limits_k \{ \max\limits_i \{ P_{ik}^{j\ell} \times U_i^{\ell\,max} \} \} \end{cases}$$

où $P_{ik}^{j\ell}$ résulte de la fusion des grandeurs P_i^{ℓ} et P_{ik}^{j} ; si l'on retient pour cette fusion la règle précedemment élaborée à partir de la Théorie de l'Evidence ou du classement statistique direct, et ceci en raison de ses propriétés en accord avec le contexte traité, il vient la détermination de :

$$(68)\qquad C_{ik}^{j\ell} = \frac{P_i^{\ell\,max} \cdot P_{ik}^{j}}{(1-P_i^{\ell\,min}) \times (1-P_{ik}^{j})}$$

qu'il convient de ramener dans l'intervalle [0, 1] par la transformation inverse :

$$(69)\qquad P_{ik}^{j\ell} = C_{ik}^{j\ell} / (1 + C_{ik}^{j\ell})$$

La Fonction d'Utilité Marginale de l'informateur j pour la situation l peut finalement être encadrée, dans l'intervalle [0, 1], par :

$$(70)\qquad \begin{cases} \Delta U_j^{\ell\,min} = (U_j^{\ell\,min} - U_o^{\ell\,max} + 1)/2 \\ \Delta U_j^{\ell\,max} = (U_j^{\ell\,max} - U_o^{\ell\,min} + 1)/2 \end{cases}$$

L'élaboration du critère Ch à optimiser pour déterminer la meilleure des H combinaisons h possibles entre les M informateurs j et les L situations l = h(j), peut alors se faire par la même technique d'inférence, pour les mêmes raisons :

$$(71)\qquad \max_h \{ C_h \} \qquad \text{où :} \quad C_h = \prod_j \Delta U_j^{h(j)max} \Big/ \prod_j (1 - \Delta U_j^{h(j)min})$$

Remarquons que, quelle que soit la méthode utilisée, l'association l = h(j) ne peut être bijective que pour L = M. Dans tous les cas, les situations non analysées par la combinaison h ne sont l'objet d'aucune modification de leur fonction d'utilité, et ne sont par conséquent pas prises en compte par (71) ou par (65)(On peut vérifier que leur prise en compte pour un informateur fictif tel que $U_j^{\ell} = U_o^{\ell}$ ne modifie le résultat dans aucun des cas) ; inversement, une situation l pour laquelle une combinaison h prévoit une analyse conjointe par plusieurs informateurs doit voir les effets de ceux-ci se cumuler, ce qui est implicitement pris en compte, aussi bien par (71) que par (65). Notons enfin qu'une gestion judicieuse des combinaisons d'intérêt par le système expert doit limiter l'aspect combinatoire des calculs à effectuer, quelle que soit la solution adoptée.

VI.3) COMPARAISON DES DEUX SOLUTIONS SUR UN EXEMPLE

Reprenons l'exemple de classification de trois cibles à l'aide de deux capteurs déja utilisé au V.2, avec les mêmes valeurs numériques. Supposons deux situations à analyser, dont chacune d'elle a déjà fait l'objet d'une investigation de la part d'un des capteurs : par convention, désignons par l = 1 la situation déjà analysée par le capteur 1 et par l = 2 celle déjà analysée par le capteur 2, quelle que soit, en fait, l'identité de la cible mise en jeu dans chaque cas. Supposons par ailleurs que nous avons deux possibilités d'associations :

- h = 1, rien ne change, chaque capteur continuant à scruter la même situation.

- h = 2, les capteurs sont inversés, le capteur 1 devant maintenant analyser la situation 2 et le capteur 2 la situation 1.

Par souci de clarté, nous retiendrons enfin : $U_i^{\ell\,min} = U_i^{\ell\,max} = 1 \;, \forall i, \ell$.

Dans ce contexte, 9 scénarios peuvent être testés, correspondant chacun à un couple de cibles différent pour le couple de situations analysées. Les P_i^{ℓ} sont dans ces conditions directement les ε_{ij} de la figure 5, pris pour l'association cible- capteur supposée réalisée. Les $P_i^{\ell\,min}$ et $P_i^{\ell\,max}$ sont bien sûr les ε_{ij}^{min} et ε_{ij}^{max} déjà élaborés au V.2 à partir des tableaux des figures 4 et 5, pris dans les mêmes conditions d'association. Les P_{ik}^{j} sont enfin directement ceux donnés par la figure 4.

La figure 10 rassemble les coûts Ch obtenus pour chaque méthode et chaque scénario, une situation étant définie par le numéro i de la cible qui la constitue. Le comportement du système après fusion, si l'on jouait l'association h préconisée, étant connu par l'analyse du V.2, il est possible d'entourer en trait continu les bonnes propositions et d'un trait discontinu les mauvaises. La comptabilisation des bonnes décisions met alors en évidence l'avantage de la méthode proposée sur cet exemple, surtout si l'on note qu'elle n'exige aucune connaissance supplémentaire ici, les P_{ik}^{j} étant de toute façon nécessaires.

VII) CONCLUSION

Un processus d'aide à la décision a été proposé, qui autorise l'emploi de différentes techniques, aussi bien pour la fusion des données mutli-informateurs, que pour l'Allocation de Ressources en temps réel, c'est-à-dire l'affectation d'informateurs à l'analyse simultanée de plusieurs situations. L'application de ces techniques à des scénarios simples de classification de cibles fait apparaître, pour chaque fonction, un net avantage de la méthode élaborée à partir de la Théorie de l'Evidence de Dempster - Shafer et confortée par une approche de classement statistique direct. Cette méthode tire l'essentiel de sa force du traitement de grandeurs incertaines, qui permet la prise en compte d'une connaissance préalable sur la qualité des différentes évaluations fournies par les informateurs. Si donc certaines méthodes offrant les mêmes possibilités, notamment celles issues de la théorie des ensembles flous, doivent éventuellement être reconsidérées dans le cadre de conditions particulières d'application, le potentiel de la méthode proposée à partir de la Théorie de l'Evidence par rapport aux approches plus traditionnelles semble un acquis indiscutable, mis en évidence par les résultats obtenus.

ANNEXE

CALCUL DE S(Hi) et P (Hi) par la Théorie de l'Evidence

Cette détermination résulte de l'agrégation de N x M, sources de connaissance caractérisant N hypothèses (d'indice i) à partir de M informateurs (d'indice j) ; ces sources sont décrites, dans le cadre de la Théorie de l'Evidence, par :

(1)
$$\begin{cases} m_{ij}(H_i) = \varepsilon_{ij}^{min} = S_{ij}(H_i) \\ m_{ij}(\overline{H}_i) = 1 - \varepsilon_{ij}^{max} = 1 - P_{ij}(H_i) \\ m_{ij}(E) = \varepsilon_{ij}^{max} - \varepsilon_{ij}^{min} = P_{ij}(H_i) - S_{ij}(H_i) \end{cases}$$

où $\overline{Hi}$ représente le complément de Hi dans E.

L'agrégation envisagée doit être menée en deux temps, pour tenir compte des particularités du problème :

1) Agrégation inter-informateurs pour les ε_{ij} concernant une même hypothèse Hi.

2) Agrégation des évaluations obtenues directement pour chacune des hypothèses Hi, afin d'obtenir l'évaluation globale de chacune d'elle.

I) AGREGATION INTER-INFORMATEURS

Cette agrégation concerne M sources de connaissance indépendantes qui décrivent le cadre de discernement E par les mêmes sous-ensembles d'hypothèses :

$$\langle m_{i1}(H_i), m_{i1}(\overline{H}_i), m_{i1}(E) \rangle$$
$$\cdots$$
$$\langle m_{ij}(H_i), m_{ij}(\overline{H}_i), m_{ij}(E) \rangle$$
$$\cdots$$
$$\langle m_{iM}(H_i), m_{iM}(\overline{H}_i), m_{iM}(E) \rangle$$

Les règles d'inférence de la Théorie de l'Evidence procurent dans ce cas :

(2) $$m_i(H_i) = \left\{ \prod_{j=1}^{M} [m_{ij}(H_i) + m_{ij}(E)] - \prod_{j=1}^{M} m_{ij}(E) \right\} / k_i$$

(3) $$m_i(E) = \left\{ \prod_{j=1}^{M} m_{ij}(E) \right\} / k_i$$

(4) $$m_i(\overline{H}_i) = \left\{ \prod_{j=1}^{M} [m_{ij}(\overline{H}_i) + m_{ij}(E)] - \prod_{j=1}^{M} m_{ij}(E) \right\} / k_i$$

avec
(6) $$k_i = \prod_{j=1}^{M} [m_{ij}(H_i) + m_{ij}(E)] + \prod_{j=1}^{M} [m_{ij}(\overline{H}_i) + m_{ij}(E)] - \prod_{j=1}^{M} m_{ij}(E)$$

Sachant que :
(7) $$P_i(H_i) = m_i(H_i) + m_i(E)$$

(2), (3) et (1) procurent :
(8) $$P_i(H_i) = \left\{ \prod_{j=1}^{M} P_{ij}(H_i) \right\} / k_i$$

où ki s'écrit d'après (6) et (1) :
(9) $$k_i = \prod_{j=1}^{M} P_{ij}(H_i) + \prod_{j=1}^{M} [1 - S_{ij}(H_i)] - \prod_{j=1}^{M} [P_{ij}(H_i) - S_{ij}(H_i)]$$

D'autre part, sachant que, par définition :
(10) $$S_i(H_i) = 1 - P_i(\overline{H}_i) = 1 - [m_i(\overline{H}_i) + m_i(E)]$$
(4), (3) et (1) amènent :
(11) $$S_i(H_i) = 1 - \left\{ \prod_{j=1}^{M} [1 - S_{ij}(H_i)] \right\} / k_i$$
où ki est toujours donné par (9)

II) AGREGATION INTER-HYPOTHESES

Cette étape prend maintenant en compte N sources de connaissance qui décrivent le cadre de discernement $E = \{H_1, \ldots H_i, \ldots H_N\}$ par des sous-ensembles d'hypothèses différents ; chaque source reste cependant de type binaire entre une proposition Hi et sa négation $\overline{Hi}$:

$$< m_1(H_1),\, m_1(\overline{H}_1),\, m_1(E) >$$
$$\cdots$$
$$< m_i(H_i),\, m_i(\overline{H}_i),\, m_i(E) >$$
$$\cdots$$
$$< m_N(H_N),\, m_N(\overline{H}_N),\, m_N(E) >$$

Ces masses se déduisent immédiatement des grandeurs Si(Hi) et Pi(Hi) déterminées à l'étape précédente par :

$$(12)\quad \begin{cases} m_i(H_i) = S_i(H_i) \\ m_i(\overline{H_i}) = 1 - P_i(H_i) \\ m_i(E) = P_i(H_i) - S_i(H_i) \end{cases}$$

Il est de plus possible d'évaluer le support et la plausibilité accordés à une hypothèse Hi par une source de connaissance voisine l ($l \neq i$) pour $N > 2$:

$$(13)\quad \begin{cases} S_\ell(H_i) = 0 \\ P_\ell(H_i) = 1 - m_\ell(H_\ell) \end{cases}$$

Compte tenu de la dépendance des différentes évaluations fournies par un même informateur, et du fait que les N sources de connaissance traitées ici résultent de l'agrégation des M mêmes informations gj, ces sources ne sont pas indépendantes. Le support global et la plausibilité globale de chaque hypothèse Hi sont donc obtenus par :

$$(14)\quad \begin{cases} S(H_i) = \max\limits_{\ell \in [1,N]} \{S_\ell(H_i)\} \\ P(H_i) = \min\limits_{\ell \in [1,N]} \{P_\ell(H_i)\} \end{cases}$$

ce qui nous conduit à :

(15) S(Hi) = Si(Hi)

De plus, les masses $m_i(\overline{H_i})$ et $m_\ell(H_\ell)$ résultant de l'inférence des mêmes informateurs, une élaboration correcte des masses initiales $m_{ij}(H_i)$ garantit $m_i(\overline{H_i}) \geq m_\ell(H_\ell), \forall \ell \neq i$, puisque $H_\ell \subset \overline{H_i}$ et par conséquent, d'après (12), (13) et (14) :

(16) P(Hi) = Pi(Hi)

Finalement, si on intègre les résultats (8) et (11) de la première étape, (15) et (16) deviennent :

$$(17)\quad S(H_i) = 1 - \left\{\prod_{j=1}^{M} [1 - S_{ij}(H_i)]\right\} / k_i$$

$$(18)\quad P(H_i) = \left\{\prod_{j=1}^{M} P_{ij}(H_i)\right\} / k_i$$

où ki est toujours défini par (9)

REFERENCES

[1] K.J. ARROW - Social choice and individual values
John Wiley and Sons Inc. 1963.
[2] D. BLAIR - R. POLLAK - La logique du choix collectif
Pour la science 1983.
[3] A. SCHARLIG - Décider sur plusieurs critères - Panorama de l'aide à la décision multi-critère - Presses Polytechniques Romandes 1985.
[4] R.L. KEENEY - H. RAIFFA - Décisions with multiple objectives : preferences and value tradeoffs
John Wiley and Sons New-York 1976.
[5] R.C. JEFFREY - The logic of decision
The University of Chicago Press, Ltd, London 1983 (2d ed.).
[6] D.J. BROWN - Aggregation of preferences
Quaterly Journal of Economics Vol. 89 n°3 1975.
[7] B. ROY - A propos de l'agrégation d'ordres complets : quelques considérations théoriques et pratiques - Colloque du CNRS sur la décision - Aix-en-Provence Juillet 1967.
[8] B. ROY - Classements et choix en présence de points de vue multiples
R.I.R.O. 2ème année n°8 1968 pp. 57-75.
[9] F.L. WRIGHT - The fusion of multisensor data -
Signal - october 1980.
[10] H. PRADE - Problèmatiques et méthodes en raisonnement approché -
AFCET - Reconnaissance des formes et intelligence artificielle 16-17 novembre 1987 - Antibes.
[11] J. LLINAS - Data fusion - Support de séminaire - SAIC 135 old Island Road - Annapolis - Maryland 21401 - USA.
[12] A.P. DEMPSTER - Upper and lower probabilities induced by a multi-valued mapping -
Annals of Mathematical Statistics n°38 - 1967.
[13] A.P. DEMPSTER - A generalization of Bayesian inference -
Journal of the Royal Statistical Society vol 30 série B - 1968.
[14] G. SHAFER - A mathematical theory of evidence -
Princeton University Press - Princeton New Jersey - 1976.

[15] J.A. BARNETT - Computational methods for a mathematical theory of evidence - Proceedings of the 7 th Joint Conference on Artificial Intelligence - Vancouver - British Columbia - Canada 1981.
[16] T.D. GARVEY - J.D. LOWRANCE - M.A. FISCHER - An inference technique for integrating knowledge from disparate sources - Proceedings of the 7 th International Conference on Artificial Intelligence - August 1981.
[17] R.A. DILLARD - Computing probability masses in rule-based systems - Technical Document 545 Naval Ocean Systems Center - San diego - CA 92152 Sept. 1982.
[18] R.A. DILLARD - Computing confidences in tactical rule-based systems by using Dempster-Shafer Theory - Technical Document 649 - Naval Ocean Systems Center - San Diego - CA 92152 Sept. 1983.
[19] L.A. ZADEH - Fuzzy sets - Information and Control n°8 - 1965 pp. 338-353.
[20] L.A. ZADEH - Fuzzy algorithms - Information and Control n°12 - 1968 pp. 94-102.
[21] L.A. ZADEH - Probability measures of fuzzy events - Journal of Mathematical Analysis and Applications vol 23 - 1968 pp. 421-427.
[22] R.E. BELLMAN - L.A. ZADEH - Decision making in a fuzzy environment - Management Science Vol 17 n°4 - December 1970.
[23] J.A. GOGUEN - L- Fuzzy sets - Journal of Mathematical Analysis and Applications vol 18 - 1967 pp. 145-174
[24] C.L. CHANG - Fuzzy topological spaces - Journal of Mathematical Analysis and Applications vol 24 - 1968 - pp. 182-190.
[25] R.C.T. LEE - Fuzzy logic and the resolution principle - Journal of the Association for Computing Machinery - January 1972 - vol 19 n°1 pp. 109-119.

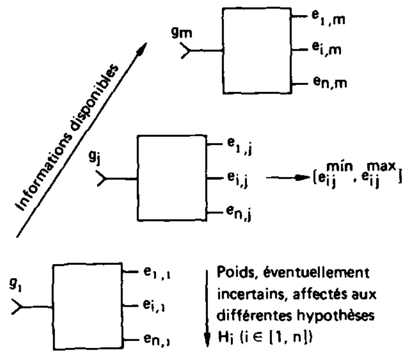

Informations disponibles
gm
$e_{1,m}$
$e_{i,m}$
$e_{n,m}$
gj
$e_{1,j}$
$e_{i,j}$
$e_{n,j}$
$[e_{ij}^{min}, e_{ij}^{max}]$
g1
$e_{1,1}$
$e_{i,1}$
$e_{n,1}$
Poids, éventuellement incertains, affectés aux différentes hypothèses H_i ($i \in [1, n]$)

Fig. 1 – Formulation du problème

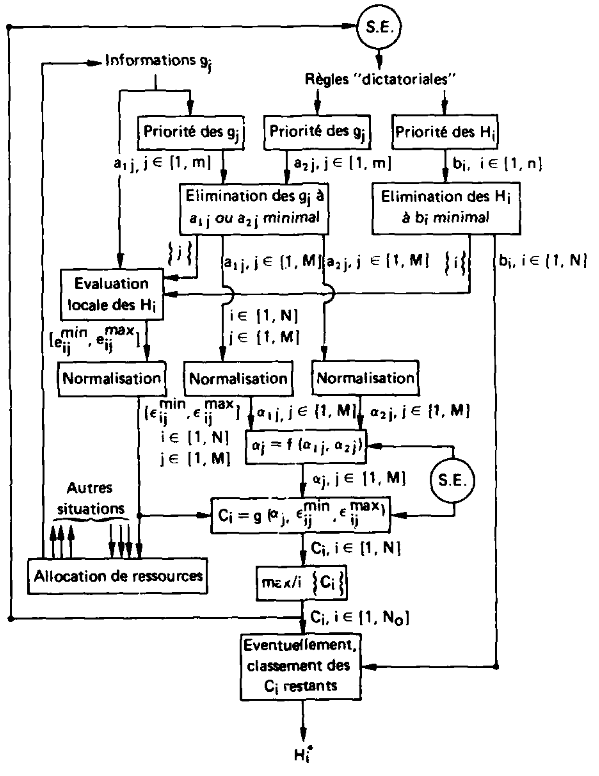


Fig. 2 – Processus de décision

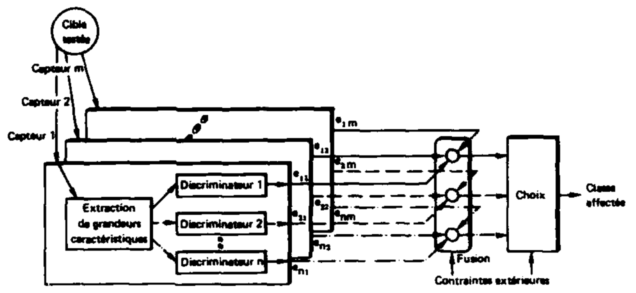


Fig. 3 – Classification multiplicateurs de cibles

P^1_{ik}		Cible présentée i =		
		1	2	3
Cible reconnue k =	1	0,60	0,15	0,20
	2	0,35	0,25	0,50
	3	0,05	0,60	0,30

a – capteur 1

P^2_{ik}		Cible présentée i =		
		1	2	3
Cible reconnue k =	1	0,35	0,45	0,20
	2	0,40	0,40	0,15
	3	0,25	0,15	0,65

b – capteur 2

Fig. 4 – Matrices de confusion

ϵ_{ij}	j = 1	j = 2
i = 1	**0,8**	0,5
i = 2	0,5	0,6
i = 3	0,1	0,3

a - Essai 1 : cible 1

ϵ_{ij}	j = 1	j = 2
i = 1	0,2	0,6
i = 2	0,3	0,5
i = 3	0,8	0,2

b - Essai 2 : cible 2

ϵ_{ij}	j = 1	j = 2
i = 1	0,1	0,3
i = 2	0,3	0,2
i = 3	0,2	**0,9**

c - Essai 3 : cible 3

Fig. 5 – Mesures à traiter

I.B.	Cible 1	Cible 2	Cible 3
C_1	**0,40**	0,12	0,03
C_2	0,30	0,15	0,06
C_3	0,03	0,16	**0,18**

M.E.	Cible 1	Cible 2	Cible 3
C_1	**– 0,525**	– 0,628	– 0,592
C_2	– 0,653	– 0,708	– 0,683
C_3	– 0,592	– 0,399	**– 0,417**

D.S.	Cible 1	Cible 2	Cible 3
C_1	**4,00**	0,38	0,05
C_2	1,50	0,71	0,11
C_3	0,05	1,00	**2,25**

Fig. 6 – Fusion des données "certaines" - Résultats

D.F.F.	Cible 1	Cible 2	Cible 3
C_1	**0,18**	0,12	0,06
C_2	0,13	0,08	0,08
C_3	0,03	0,13	0,06

D.C.F.	Cible 1	Cible 2	Cible 3
C_1	**0,59**	0,43	0,35
C_2	0,52	**0,48**	0,42
C_3	0,38	0,45	**0,59**

D.S.	Cible 1	Cible 2	Cible 3
C_1	**1,69**	0,64	0,41
C_2	1,11	**0,89**	0,66
C_3	0,51	0,68	**1,82**

Fig. 7 – Fusion des données "incertaines" - Résultats

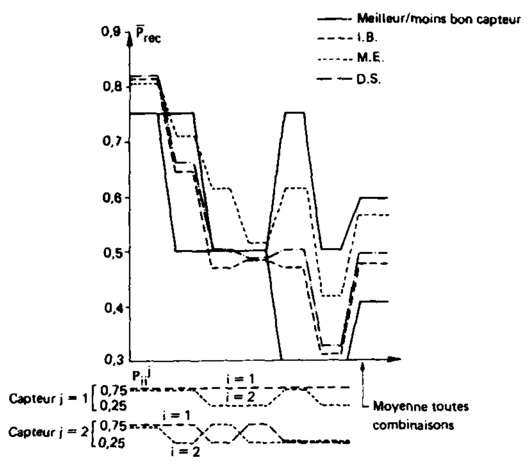


Fig. 8 – Etude statistiques en données "certaines"

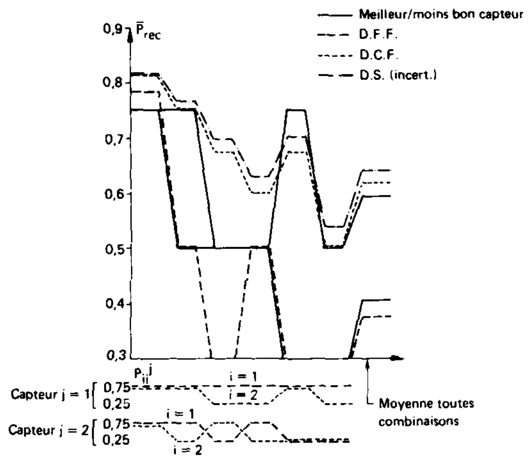


Fig. 9 – Etudes statistiques des données "incertaines"

Situation		Solution bayésienne		Solution proposée	
j = 1	j = 2	h = 1	h = 2	h = 1	h = 2
i = 1	i = 2	0,28	0,27	0,70	0,75
i = 2	i = 1	0,11	0,24	0,87	0,91
i = 1	i = 3	0,27	0	0,68	0,62
i = 3	i = 1	0,135	0,140	1,01	1,03
i = 2	i = 3	0,01	0	0,71	0,74
i = 3	i = 2	0,05	0,36	0,97	1,01
i = 1	i = 1	0,37	0,24	0,84	0,76
i = 2	i = 2	0,02	0,27	0,83	0,90
i = 3	i = 3	0,04	0,09	0,827	0,829
Bonnes décisions		6/9		8/9	

Fig. 10 – Allocation de ressources -Résultats

Role and design of a Software Engineering Data Base in the context of an Advanced Support Environment.

Michel Lemoine

ONERA-CERT
Département d'Etudes et de Recherches en Informatique
2, avenue E. Belin
31055 Toulouse CEDEX France
e-mail : lemoine@tls-cs.UUCP

Laurent Pommier

Ecole Nationale Supérieure de l'Aéronautique et de l'Espace
10, avenue Edouard Belin
31400 Toulouse CEDEX France

Abstract

This paper addresses the role and the design of a Software Engineering Data Base in an Advanced Support Environment. A first introduction recalls the evolution of the File Management System, from the earlier beginning of the Operating System up to now where sophisticated ones are available. It is shown why the classical File Management System is not sufficient in the context of the software development process. Thus Object Management Systems are now under consideration. We examine their role and for some Software Engineering Environments we exhibit what are their main functionalities. But because they are independent on the languages they support, they work at the macroscopic semantics of the manipulated objects. Furthermore, they only concern static (or predefined) properties of the development process. Thus there is great need for a Software Engineering Data Base in which more semantics can be taken into account, especially at the level of the object description. This will allow to guarantee the consistency of the information stored in a Project Data Base.

This project is supported by the ESPRIT Programme (#P510 ToolUse) and ONERA.

Keywords: *File Management System, Object Management System, Project Data Base, Software Engineering Data Base.*

Mots-clés: *Système de Gestion de Fichiers, Système de Gestion d'Objets, Base de Données Projet, Base de Données de Génie Logiciel.*

1 Introduction

Software development requires today more sophisticated supports than File Management Systems (FMS) could offer. Indeed an FMS offers a poor universe since it only manipulates *files*. The use of an FMS from a programming language like Fortran obliges the user to describe everything about the internal structure of files. This remark leads us to consider abstractions of files. Now we may introduce a first definition of what we will call *objects*:

A software object is any basic information manipulated by the programming environment.

According to this definition, every instance of a file in the FMS is a particular object. We mean that every program has its own view of files. Tools manage their own representation of the information generally ignoring all other points of view. This basic structure is not sufficient for Advanced Support Environments (ASE) because it doesn't allow processors to cooperate in a simple way.

2 From FMS to OMS

In the FMS supported by UNIX, the notion of object is not commonly shared by the environment. Nevertheless UNIX offers a good basis for the construction of Software Engineering Data Base (SEDB). We mean that the FMS of UNIX tends to *unify the concept of file* since it treats every entity as a file (classical files but peripherals as well).
Moreover some commands allow us to extract some additional information about files. Let us look carefully at the *file* command. *file* lists details about the type of files. We could call this information an *attribute* attached to a file even thought it is not explicitly recorded. This approach is more obvious if we consider UNIX tools such as SCCS which manipulates more structured objects than FMS does. Indeed in SCCS, objects are enriched by attributes which allow the tool to *know* elaborated information about them. But this knowledge remains private to SCCS.

Now let us look at a step forward. An ASE needs a *centralized* data base whose management system manipulates sophisticated objects. Such systems are usually called Object Management Systems or OMS.

An OMS can be considered as a Data Base Management System (DBMS) having some knowledge about the objects it manages. An OMS has several roles:

1. It will be responsible for all the problems of communication between the Data Base (where objects are stored) and a working space in which they will be manipulated, transformed, and so on.

2. Because the OMS has some knowledge about the objects it manages, it would be able to support specific actions beyond the classical management aspect. For instance, it would be responsible for the *consistency checking* before deleting, updating or creating any object. In this case, consistency may not be assimilated as Data Base consistency checked by the DB model but as a syntactic and semantic control.
   Let us simply consider the EMACS text editor. Before you stop a session without saving your text, EMACS asks you if you want to save it. This basic action is done by the tool itself in a classical FMS environment but should be taken into account by the OMS in an ASE. In other words, an OMS must offer the means to express some integrity rules, consistency checks, and so on.

The Portable Common Tool Environment (PCTE) may be seen as an evolution of UNIX-FMS towards this aim. In this system, several kinds of objects are considered. PCTE which is built on top of an FMS , introduces the concept of Object Management System. But it doesn't fulfil all the requirements of an OMS. Indeed the OMS of PCTE doesn't carry *semantic information* about the objects.

3 Several kinds of OMS

Several ASE under development include an Object Management System. These ASE use data models which lead to a common definition of what should be an OMS.

3.1 The PCTE environment

From our point of view, PCTE described in [PCT85] is an intermediate step between FMS and OMS. In the PCTE model we can extract:

- **objects:** they correspond to the traditional UNIX file concept.
- **attributes:** they represent some additional values attached to objects.
- **relations:** they express the logical associations and dependencies between objects.

In fact, the OMS of PCTE includes the SCCS functionalities and moreover collect a lot of information about software development. On the other hand the OMS of PCTE doesn't manage any semantic information on objects. More precisely PCTE allows the user to link various representations or versions of objects. These objects are referred through pathnames (like UNIX ones) whose PCTE checks the syntax according to derivation rules.

3.2 The Project Master Data Base: PMDB

The PMDB system described in [PS85] aims at the integration of a method for the entire Software Life Cycle. The PMDB distinguishes the following concepts (close to the PCTE ones):

- **objects:** components which characterize the type of the data which are relevant to a project. They are seen as sets of data which have same properties (software components, resources, dictionary for example). Objects might be instanciated with specific values.
- **attributes:** they express the characteristics of objects. Every instance of an object has proper attributes.
- **relations:** they represent associations between objects. The links between instances of objects obey to the relations between these objects. We mean that if a user wants to link two instances of objects then these objects must be related.

In this approach the model represents only a schema which has as main drawback to be static. The project skeleton is fixed by the structure of the objects and the relations between them but it can't evolve during the transactions. PMDB only checks the syntax of the project structure. But except elementary type checking, it doesn't carry any significant semantics.

3.3 The SPRAC system

A significant example of the functionalities of an OMS is shown in SPRAC [FJL*85a] and [FJL*85b]. SPRAC aims at the integration of three consecutive steps of the Software Life Cycle (specification, design and implementation levels). The data model of SPRAC is organized around three concepts:

- **objects:** there are three kinds of objects corresponding to three kinds of the software components (functions and abstract data types, algorithms, modules).
- **relations:** there are two kinds of relations:
  - the explicit relations correspond to classical links between the objects such as *uses, is_represented_by, realizes ....*
  - the computed relations are obtained by a calculus on the explicit relations. For instance:
    The GDD relation corresponds to *what has been done*.
    The AGENDA relation corresponds to *what should be done* according to a predefined method.
- **semantic actions:** actions associated with integrity rules (constraints). They express the properties that the relations must satisfy and consequently maintain the consistency of the Project Data Base.

The SPRAC system treats semantics (of the supported languages) but the semantic actions are not part of the data base. The latter look for language elements inside objects and checks their consistency with the other objects. The OMS is not well discerned in SPRAC even thought it is underlying.

3.4 The GRASPIN Data Base

The model shown in [GSZ*87] is based on the object oriented approach. Consequently it is articulated around the notion of *classes*. A class is decomposed into:

- A *data structure* which corresponds to the implementation of the object structure.
- A set of *manipulation methods* which express the access interface of the object.

In GRASPIN, the data base schema is defined by the description of the classes called *Language Environment* or LE. Each LE is a description support for a specific user's language. These classes are created from a syntactic and semantic specification of the language.

The main advantage of this approach is that the OMS has a deep semantic knowledge of objects it manipulates. By example it can store and update an Abstract Syntax Tree with semantic attributes. The RELATION and TREE system classes provide the means to evaluate and propagate these attributes in the tree. So tools like compilers or syntax directed editors or specialized checkers can be built above GRASPIN in an easy way.

Another interesting particularity lies in the object oriented interface because it hides all the implementation details to the user.

The GRASPIN DB allows extensibility of the object definitions. Moreover, due to its semantics knowledge, its model is dynamic because it can evolve (relations are created) during a session.

4 Definition of the OMS concept

A formal definition of what might be an OMS has been given in [Mey85]. This definition is not enough concerned by the semantic aspects supported by the objects themselves. Indeed the proposed model shows us how the relations between objects can be handled and implemented but not why (semantic aspects) they have been chosen.

In [Kra82] the author proposes a schematic model constituted of objects, relations and operations. This model matches with what we mean by OMS. Nevertheless the model addresses only the syntactic aspects of the objects.

Thus we suggest the following model to which syntax and semantics can be attached:

1. **objects:** objects are basic entities which compose any software project. For instance *program module, requirements, documentation* are software objects. They are not supposed to be atomic since they can be decomposed. By example a part of program module (such as a *procedure*) can be considered as an object itself. Objects are (classically) decomposed into:

   - *content:* it supports information which is significant for the user and (unfortunately) not for the OMS.
   - *attributes:* they represent some additional information about the object. This allows the OMS to *know* more about it. The attributes support part of syntax and/or semantics.

   An object may be instanciated. This means that the user may introduce some instances of objects with specific values of attributes.

   example: Let us look at *program modules* in block structured programming languages. First we create the object *program module* in our OMS.

   - Its attributes are *name, type, size, programming language, time of last change ....*
   - Its content is *the text of the module.*

   Now we consider an instance of this object. The function *LOG* returns the logarithm of a real number.

   - The values of its attributes are *LOG* for the name, *function* for the type, *500 characters* for the size, *source Pascal* for the programming language, *1/1/88* for the time of last change ....
   - The content is the source code in Pascal of the function LOG.

2. **relations:** they constitute the typed dependency links between the objects. They contribute both to the syntax and semantics of the software project.

   example: The *is_used* relation links the LOG object to all the objects in which LOG is used.

3. **semantic operations:**

   - *constraints:* they define the validity conditions on relations and attributes.
   - *actions:* they are the procedures to be applied when constraints are violated.

Operations are the support for the deep semantics of the software project in the OMS. They provide the mean to ensure the consistency of software developments.

example: If we suppose the constraint on any declaration is: *only backward reference can exist* then we can imagine the action *forward_reference*. Whenever the user introduces a module reference the constraint is checked. If the constraint is satisfied then a tuple of the *is_used* relation is created. In case of failure the *forward_ reference* action is performed.

5 Towards more semantics managed by OMS

We now consider in more details some examples of constraints (and the associated semantic actions) which do exist in some Software Engineering Environments.

5.1 Integrity rules in SPRAC

In the SPRAC system, there are several kinds of consistency checks e.g. semantic operations [FJL*85b]. Their performance ensures the consistency of the Project Data Base (PDB) in which all the software objects are built and managed.

A first kind of checks is done during the compilation phase, each time a given object is introduced. For instance, any object may refer to other objects. The consistency check corresponding to the *import clause* consists in verifying that:

- If the referred objects exist in the PDB, then they are consistent with their use. Consistent means that *name, arguments* and *types, interfaces*, ... are valid both at the level of the use and at the level of the declaration. If the test fails then the introduction of the object is rejected.
- If the referred objects do not exist in the PDB, then no checks are made.A predefinition of the referred object is generated and put in the PDB. The consistency check will be done when the definition of the forward reference will be completed later on.

Another example of integrity rule is the introduction of *laws* corresponding to part of a method the programmers have to follow. In SPRAC, there must exist for any module both a specification and an algorithm. A law corresponding to a top down development may state:

A specification must be done before the creation of an algorithm
and
an algorithm must exist before writting a program

This two laws are interpreted as integrity rules by SPRAC when a new object is given to the system. The application of the two integrity rules may conduct to the rejection of the object if the laws are not satisfied. By this way we ensure the consistency of the PDB with respect to a method.

The model of the OMS supported by SPRAC allows to make such consistency checks. But the consistency checks are isolated in many independent functions called each time there is a need for them. No real integration of the semantics depending on the language in which objects are described is taken into account by the OMS of SPRAC.

5.2 Integrity rules in B

The B system [Abr85] is a general proof checker allowing to create,delete, modify formulas and to handle them with the help of a metalogic. One of the main interesting capabilities of B is its ability to help users create proofs (in the mathematical sense). On one hand we have formulas organized as "theories" (in which formulas represent either axioms or theorems), on the other hand we have proofs. In B, proofs are structured objects. They are *trees* in which:

- *nodes* support the transformed formula.
- *arrows* contain either a reference to an applied transformation (like GENERALISATION, DEDUCTION, HYPOTHESIS, ...) or a REWRITING rule (axiom or theorem) used for the transformation.

Some examples of integrity rules (corresponding to what is usual in mathematics) are:

- *each time a proof is under development, no modification in the used theories are available.* In other words the proof activity restricts the capability of the B system from the point of view of the DB aspects.
- *if any formula (axiom or theorem) is removed then all the proofs using it are removed.* This is a very strong integrity rule because it can be followed by the deletion of a lot of proofs. A less restrictive one could be: *if any formula is removed then all the proof using it are to be replayed.* In the latter the system warns the user to pay attention to the validity of some proofs.

5.3 Need for OMS supporting integrity rules

The main conclusion from these examples of constraints (and consequent actions) is that integrity rules do really exist but are performed as separate functions. Even if they correspond to a single semantic actions, they are not gathered in a unique component.

This remark can be extended to programming languages ranging from FORTRAN to ADA. Indeed if we look carefully at these languages, a problem such as modularity, clearly shows that consistency checks are needed. Depending on the maturity of the considered language, different levels of checks should be available.

For instance in FORTRAN no checks at all are made, neither at the compilation nor at the run. On the contrary in ADA some checks are performed. For instance, the *USE clause* refers to objects (data or programs or packages) to be imported in the module where the clause appears. But as for SPRAC, checks are made at different times in separate functions.

The main reason is there doesn't exist real SEDB but only semantic functions from place to place and used in specific parts of the system.

A SEDB should act at a microscopic level of objects which is the level of the semantics supported by the language in which objects are described. In GRASPIN [GSZ*87] this requirement has been partially taken into account.

6 An example of SEDB in the context of TOOLUSE

We now introduce a Software Engineering Data Base in the context of an under development ASE. We first examine very briefly the role of the ASE, then the objects it manages and the model we have developed.

6.1 The TOOLUSE project and DEVA, a development language

The TOOLUSE project [Hor85] is the continuation of the SPRAC project. TOOLUSE emphasizes the role of methods in ASE. In other words, it aims at producing the means to express software development descriptions. A development of any piece of software can be made with the help of the development language called DEVA [SWdGC87]. DEVA is a wide spectrum language which allows to describe all the components built and used during the software life cycle. Among them: specifications, developments and programs. The availability of DEVA as development language is the first step for consideration of methods in ASE. Indeed, using DEVA makes consider methods e.g. the ways software must be developed according to specific recommendations (target system, application area, company rules).
The structure of a DEVA program is of the following forms:

- $(A \vdash B)$ which can be interpreted as "from A we can derive B" (inference rule) or *the program B satisfies the specification A* (development process). Both interpretations are valid.
- $((A \vdash B) \dashv C)$ which is a more general form and can be interpreted as *from A we can derived B according to the proof C* or *the development C ensures that the program B satisfies the specification A.*

In this paper, we are not considering DEVA in details. We will only exhibit how around these general structures $((A \vdash B)$ or $((A \vdash B) \dashv C)$ a SEDB has been built.

6.2 The DEVA SEDB

The semantic constraints (other than those ones ensured by classical DBMS) are strongly related to those ones from SPRAC and B. Indeed main of the objects manipulated with DEVA inherit from SPRAC (software objects) and from B (proof construction). A proof (in the sense of B) is nothing more than a development (in the software development process terminology).

6.2.1 The SEDB model

The model we have chosen corresponds to the definition previously given. The instance of this model is based on the relational approach. More precisely, we restrict our (instance of) model to binary relations:

1. *is-object (ident, formula)*
   which makes the link between a definition and an identifier.
2. *has-type (ident, type-value)*
   which attributes a type to any DEVA object: specification, program, proof, law, rewriting rule, ADT, function, ...
3. *is-certified (ident, boolean)*
   which enables the system to guarantee an object is valid or not.
4. *is-constructed-by (ident, ident)*
   which represents the DEVA schema $(A \vdash B)$ with its meanings. The general schema $((A \vdash B) \dashv C)$ is ternary. Thus it needs to be decomposed into:
   is-object (id1,$(A \vdash B)$*)*
   is-object (id2,C)
   is-constructed (id2,id1).

5. *is-antecedent-of (ident, ident)*
   which gives the list of direct antecedent of the 1st argument.

6. ......

This model derived from the definition of the manipulated objects is static. The tuples of relations are instanciated at the definition time (edition or compilation).

Because the model is directly implemented through binary relations, it is possible to make it dynamic. By dynamic we mean the availability to introduce new objects and/or new relations. Indeed, in our approach, there doesn't exist an OMS schema but only an instanciation. This advantage is unfortunately counterbalanced by the need of semantic actions applied to the relations themselves.

6.2.2 The SEDB as a main component of TOOLUSE

The architecture of the DEVA support environment is as represented in figure 1.

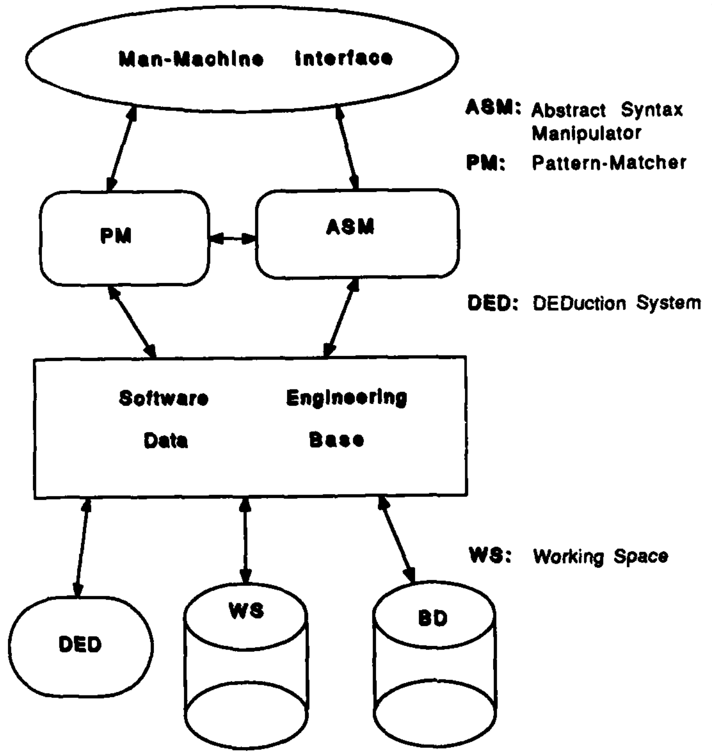


figure 1

The components are:

- the MMI which enables the conversion from external syntax to internal one (manipulated by the ASM). During the transformation, the syntax of the DEVA language is used to generate checks by calls to the SEDB. For instance as soon as an object is referred then its definition (of course if it exists) is shown to the user. In the same time checks are made about the validity of the call.

- PM is a pattern-matcher used for the retrieval of information (objects) either in the Data Base (BD) or in the Working Space (WS). Any reference to an object will be translated to a call to the SEDB which in this case performs the call as a classical DBMS.
- DED is the deduction system which allows to run DEVA programs. Attached to it are a lot of semantic actions only performed at the run time.

In this architecture, the first role of the SEDB is a DBMS role e.g. it is called any time an operation needs the presence of more information not directly available within the operation.

The second role is concerned with the semantics any operation may require. For instance, if an operation (supported by DED) consists of replaying a development in a slightly modified context (for a yet developed program) then the SEDB will be responsible for checking the availability of the replay and in case of success for the certification of the new development.

7 Conclusion

This paper presents the role of an Object Management System in the context of Advanced Support Environment. As shown through several ASE under development the OMS will constitute the kernel of the system, both from the viewpoint of its DBMS role and from the viewpoint of the integrity of the software development.

For this purpose we suggest to regard a SEDB as supporting as much as possible semantics of:

- the objects manipulated. The semantics can be deduced from the description language used. It is important to notice that a domain application language will conduct to more semantics and therefore to more constraints.
- the underlying model of software development. It will be necessary to construct (by hand) some specific relations and to develop corresponding semantic actions.

The model we have developed carries out the precedent objectives. The experimentations are conducted in the context of an ASE which is quite devoted to the formalization of software development in order to replay some existing developments and to adapt them to new contexts. By this way, we hope to be able to favor the reuse of software.

References

[Abr85] J. R. Abrial. *The B project.* Technical Report, ADI, Contract 85C00410, 1985.

[FJL*85a] J. Foisseau, R. Jacquart, M. Lemaitre, M. Lemoine, and G. Zanon. Le système SPRAC : expression et gestion de spécifications, d'algorithmes et de représentations. *TSI,* 4(3):237–254, 1985.

[FJL*85b] J. Foisseau, R. Jacquart, M. Lemaitre, M. Lemoine, and G. Zanon. SPRAC: from Specification to Programs. In J. MacDermid, editor, *Integrated Project Support Environment Conference*, pages 114–123, IEE-SES-1, 1985.

[GSZ*87] S. Goutas, P. Soupos, C. Zaroliagis, D. Christodoulakis, and D. Maritsas. The use of the object-oriented approach in the GRASPIN DB. In CEC-GDXIII, editor, *ESPRIT'85: Status Report*, pages 361–374, North-Holland, 1987.

[Hor85] H. Horgen. TOOLUSE: an advanced support environment for method-driven development and evolution of packaged software. In CEC-GDXIII, editor, *ESPRIT'85: Status Report*, pages 545–562, North-Holland, 1985.

[Kra82] S. Krakowiak. Systèmes intégrés de production de logiciel : concepts et réalisations. *TSI,* 1(3):187–200, 1982.

[Mey85] B. Meyer. The software knowledge base. In *Proceedings of the 8th ICSE*, Imperial College, London (UK), 1985.

[PCT85] PCTE team. An overview of PCTE: a basis for a portable common tool environment. In CEC-GDXIII, editor, *ESPRIT'85: Status Report*, pages 313–329, North-Holland, 1985.

[PS85] M. H. Penedo and E. Don Stuckle. PMDB : a project master database for software engineering environments. In *Proceedings of the 8th ICSE*, Imperial College, London (UK), 1985.

[SWdGC87] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. *Definition 0.1 of the approximation DEVA.0 of a development language.* Technical Report, UCL, TooIUse.TD.deva01.DD88a, 1987.

DISCUSSION

A.O.Ward, UK

In your REPLAY system are you replaying the human design decision or the formal transformational steps (as happens with REFINE)?

Author's Reply

In the REPLAY system, the sequence of transformational steps recorded during a development is replayed. DEVA, the tool development language, formalizes the development by recording what has been done during the development but it has no way to formalize why one transformation was better than another. This is the role of what is called a CONTEXT.

ADA IN EMBEDDED AVIONIC SYSTEMS

W. Mansel
Messerschmitt Bölkow Blohm GmbH, Military Aircraft Division
Postfach 80 11 60, D-8000 München 80, West Germany

Summary

Ada will be used for real-time avionic software on the TORNADO aircraft upgrade programme and on the EFA programme. For the HARM integration programme onto the TORNADO aircraft an Ada crosscompiler based on the Karlsruhe compiler technology configured to a multiprocessor avionic computer is used. Software prototyping has been applied to gain experience with Ada avionics application. The results will be discussed as well as the experience from benchmark tests on different Ada crosscompiler systems for single processor MC 68000 and 68020 microprocessor targets. Special emphasis will be taken on Ada tasking and the parallelism in avionic software.

1. Introduction

The avionic systems of modern military aircrafts are becoming more and more complex. This is due to the increasing complexity of expected battle field scenarios and consequently increasing complexity of tactical requirements. To meet that requirements for increased computing capacity and amount of stored software,distributed avionic systems have been developed. The individual computers in the onboard network have dedicated tasks as e.g. navigation, cockpit management or weapon management.

The characteristic features of an avionic computer are 1 - 5 MByte memory capacity and 1 - 3 MOPS processing capacity. Whereas that has been achieved usually by bit slice machines there is a strong trend especially in Europe to use standard 16 or 32 Bit microprocessors. To achieve the apropriate processing throughput,new avionic onboard computers are multiprocessor architectures with 2 to 4 microprocessors and additional I/O processors for e.g. STANAG 3838 bus (MIL 1553) handling.

From the use of standard microprocessors there is a major benefit for the development of avionic software. For the development off-the-shelf tools as macro assemblers or compilers for a large variety of high order languages and software development environments are available. A similar situation exists for software testing using debuggers and development systems including emulators.

With growing complexity and amount of avionic software there has been the requirement from the customer to the avionic industry to use Ada as standard high order language for the implementation of embedded systems software, to increase the reliability, maintainability and reusability, to lead to a long term reduction of development costs. This paper discusses some aspects of software development for multiprocessor avionic computers with the special emphasis on Ada and Ada compiler systems.

2. Software Partitioning and Tasking Strategies for Multiprocessor Architectures

The software partitioning and tasking strategies can not be discussed independently from the hardware architectures. Three typical architectures are shown in Fig. 1. Example b and c where the processors have local memory but interprocessor communication is achieved by shared memories are typical for avionic computers. Dedicated input/output (I/O) processors must have access to data in shared memories. As a consequence for architecture b one of the processors has to be coupled to the I/O processors resulting in a master-slave configuration.

The software partitioning has the aim to achieve an optimum distribution of processor loads. There are two general strategies:

a) The software is designed without regard to the target architecture. It is partitioned after it has been written. This so called post-partitioning does not consider any hardware or software architectural aspects during the design.
In this case a language is required to express the partitioning of the programme onto the target. In addition the run-time cost of partitioning is hidden and time constraints can not be guaranteed.

b) The software is designed with the distribution in mind. For that socalled pre-partitioning a selected programme construct (e.g. Ada library unit) is the unit of partitioning. The designer has to obey certain restrictions in inter-processor units.

During the partitioning the software has to be separated into parts which have to be processed in sequence (modules) and parts which can be processed in parallel (tasks). One module has to be allocated to one processor, while tasks are candidates to run in parallel on different processors. However minimising the communication and data exchange between the different processors is a mandatory rule. The same rule is also valid for the total number of tasks which should be at a minimum. This is a contradictionary requirement for real-time systems, where reduction of tasks in general result in stronger dependencies of the software parts.

Also the timing and synchronisation between software parts running on different processors have carefully to be considered. The access to I/O lines is essential for embedded software. Since for the multiprocessor architecture b (Fig. 1b) one processor (master) is connected to the I/O processors, the complete I/O handling has to be centralised on the master while the slaves should be used for data processing.

The interprocessor synchronisation of software partitions is of central interest for multiprocessor avionic computers in real-time application in embedded systems. This synchronisation is realised by using the relevant utility functions of real-time executive kernels e.g. the Ada run-time system.

The traditional approach is to use a rigid scheduling schema on every processor. This fully cyclic solution reduces the number of tasks on one processor to the number of time slots in the system and results in a deterministic timing behaviour on one processor and is easy to test. But this has to be payed with a more complex software structure since e.g. exeption handling due to passed over timeframes has to be implemented. In addition special packages for the interprocessor data exchange and synchronisation have to be developed. In general the rigid scheduling schema will result in a rigid software structure which is more difficult to adapt to later requirements and therefore unfrinedly for maintainance. This approach is only convenient for small, dedicated systems.

A more adequate approach for complex real time embedded systems is a totally event and priority driven system. This asynchroneous solution which is strongly supported by Ada implements independent processes as tasks which communicate and synchronise by the Ada rendezvous. The task scheduling is ruled by the task priorities. In general the number of tasks in a fully concurrent software is larger as for rigid scheduling. The system is more difficult to test. However the software system is much more flexible to later changes to implement new requirements. It is easier to maintain. Possible timing problems due to the increased number of tasks can be solved by appropriate adjustments of task priorities into high priority tasks for the time critical parts and low priority tasks. An event and priority driven system also fits well onto a multiprocessor target, since the inter-task communication handled by the run-time system provides suitable mechanisms for interprocessor communication and synchronisation.

On the other hand, the interfaced system devices must be flexible enough, to accept a time jitter with no loss of performance, so that in a multitasking system the little time-penalty due to contex switch of tasks (200 usec) are tolerable.

3. Ada Compilers for Multiprocessor Targets

The options for distributing a software package implemented in Ada on to multiprocessor targets range between

- one separate Ada programme for each processor

- one single Ada programme distributed across the processors.

The first model can be realised using off-the-shelf Ada crosscompiler systems. However in that case no explicite Ada mechanism is available for interprocessor communication. Such a model therefore requires the programmes to communicate with each other via the facilities provided in Ada for normal input/output devices. For embedded targets considered here, this would be through low level input/output, or even through assembler code insertions. The disadvantages of such an approach are that the software is inflexible against changes in the software partition which reduces reusability and maintainability. In addition the compiler and the language tools can not be used to check the interfaces between the programmes.

The use of off-the-shelf Ada crosscompiler systems needs customisation to the target with respect to clock and time handling, host-target communication and protocol, target resident loader and debug support and import of foreign object code modules.

The second model which requires a real Ada multiprocessor crosscompiler has the main advantages that

- interprocessor communication can be implemented as inter-task communication via the Ada rendezvous. By that a unique task communication mechanism provided by the Ada compiler can be used on the multiprocessor target.

- the interfaces between the software components on the individual processors can be checked with the Ada system.

As a result the software structure becomes more flexible for redistribution onto the processors of a multiprocessor target which strongly increases the reusability and maintainability which has to be payed by a more complex Ada system.

The general features of a multiprocessor Ada system are shown in Fig. 2. Besides the extension of the Ada run-time system to handle a interprocesser rendezvous, the essential part of this is the builder tool. The builder has to check the dependencies of the programme components and establishes from the description of the target architecture the programme mapping, which is the basic information to be used by the loader and the target debugger.

4. The TORNADO Ada System for HARM Integration

This chapter discribes an Ada System developed for the HARM integration programme onto TORNADO. The target computer is a Motorola MC 68000 based multiprocessor architecture of the master-slave type as shown in Fig. 3. To fully use the benefits of software written in Ada with respect to flexibility in software partitioning the requirement has been placed onto the Ada compiler to support the multiprocessor target structure and allow the distribution of a single Ada programme onto the target.

4.1. The Ada Crosscompiler System

The resulting Ada cross system developed by Systems Designers (TAS-2) comprises of the following components (Fig. 4):

- ANSI Ada Cross Compiler running on VAX/VMS

- Optimized Runtime System with real time performance allowing intra-processor and inter-processor rendezvous

- Builder/Linker/Loader System compising:
  Definition Tool to specify the target configuration and the partitioning of the software;
  Aquisition Tool for the inclusion of assembly code units;
  Builder Tool for linking and allocation of code to memories sections
  Loader Tool for downloading software via RS 232 line

  Target Debugger providing a user-interface and functions for symbolic debugging

The Ada crosscompiler includes the following features:

- Optimized Ada tasking implementation with priority-based preemptive task scheduling

- Message passing package

- Set time and date function

- Support of MC 68881 coprocessor

- Mathematic packages providing
  - floating point functions
  - fixed point functions

- Input/Output packages
  - character and string I/O
  - discretes I/O
  - STANAG 3838 bus I/O

4.2. The Target Specific Packages

The target specific I/O packages are a special feature of the TAS-2 Ada system. This packages however are crucial for embedded systems, but usually are not provided by off-the-shelf Ada target systems and have to be developed by the user.

Most I/O facilities in avionic computers are serving interrupts and therefore need an interface to the run-time system, but on the other hand also interface to the firmware which handles the I/O driver chip sets. Due to that interrelations it has been decided for the TAS-2 Ada system to implement the I/O functions as run-time library support packages with assembler boddies for efficiency reasons, providing a high level Ada interface to the user. This strategy can also be extended to other firmware functions to be used by the application software. The standardisation of the Ada interface to the user of target specific packages introduces a higher degree of abstraction into the software for embedded avionic systems, thus again increasing reusability and maintainability. This strategy has also been selected for Ada target systems to be used within the EFA project.

4.3. Architectural Dependencies

The development of Ada compilers which fully support multiprocessor architectures has great advantages from the user point of view, as has been discussed. However, what are the limitations when trying to adapt such a product to other multiprocessor systems?

TAS-2 was designed for a special multiprocessor target, shown in Fig. 3. However it was a design goal to keep the architecture dependencies to a minimum.

The class of multiprocessor architectures TAS-2 supports can be characterised by the following requirements:

a) All processors must be of the same type.
b) There must be a shared memory between any pair of processors that need to communicate.
c) There must be an interrupt facility between any pair of processors that need to communicate. Interrupts must be possible in both directions.
d) The shared memory must be seen at the same address from the sharing processors.
e) There must be a (large enough portion of) local memory which is seen at the same

address from the various processors (a sufficient condition is that the local memories of each processor are seen at the same address).

f) A single shared memory address must correspond to only one memory location across the entire system.

For examples that fall into this class of architectures, see Fig. 1.

5. Ada Software Structure for a Multiprocessor Target

A typical example of the operational software for an embedded avionic system for a 3-processor target of master-slave structure (extension of Fig. 3 to 2 slaves) is outlined in Fig. 5. The software is structured into four major categories of software packages: Packages for I/O data handling, control packages and calculation packages with high and medium priorities and low priority background packages.

The operations imported from an I/O-package enter the data from pilot control pannels or from sensors via a STANAG 3838 bus. Depending on the data entered from the pilot control pannels control package operations select the function imported from the calculation package to be processed. The processed data are transferred via the STANAG 3838 bus to other subsystems. As can be seen from Fig. 5 the I/O packages, the control packages and the calculation packages represent parallel tasks on different processors of the multiprocessor avionic computer.

In the event that this model of an avionic software is implemented in Ada, all tasks will communicate in terms of the Ada rendezvous. Since avionic software is time critical, it can not be tolerated that any controlling task is blocked as a consequence of its entry call to an associated calculation task until the rendezvous with this task is completed. Therefore the Ada rendezvous concept enforces the introduction of additional buffer tasks for deblocking the control and the processing tasks, as can be seen from Fig. 5.

6. Conclusions

Several aspects of software design strategies for multiprocessor avionic computers have been discussed with the special emphasis on Ada and Ada compilers. It has been shown that there are encouraging arguments for the development of multiprocessor Ada systems. The use of such Ada crosscompiler systems for the development of avionic software is beneficial to exploit the concepts inherent in Ada and to achieve the expected long term improvement in maintainability and reusability. That arguments seem to justify the additional development effort in excess to the use of off-the-shelf Ada crosscompiler systems for single processor targets.

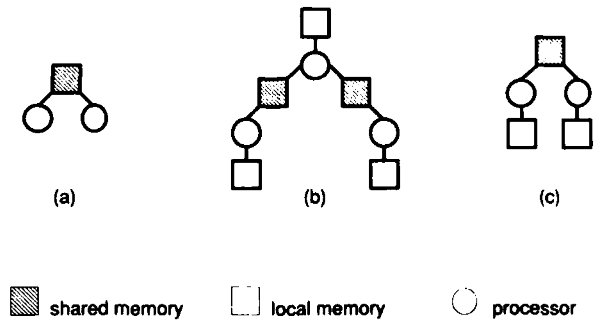


Fig. 1: Three typical multiprocessor architectures

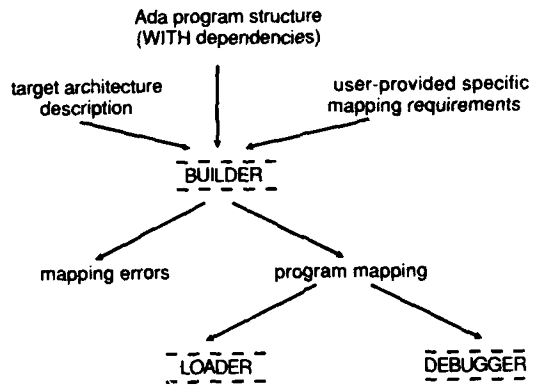


Fig. 2: Functional structure of a multiprocessor Ada compiler system

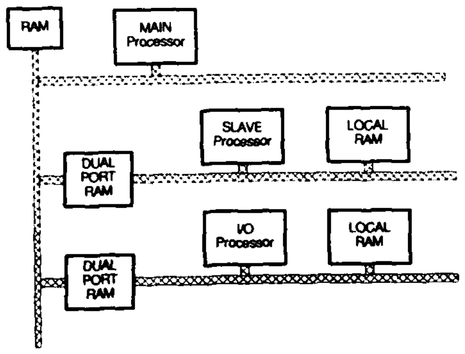


Fig. 3: Hardware architecture of the avionic computer for HARM integration programme

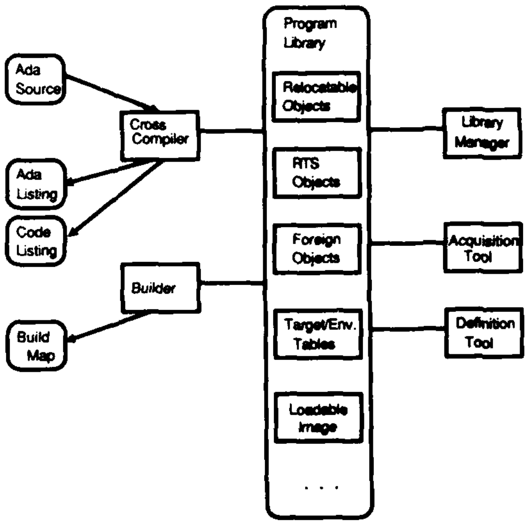


Fig. 4a: Ada cross development environment for the *HARM integration programme*

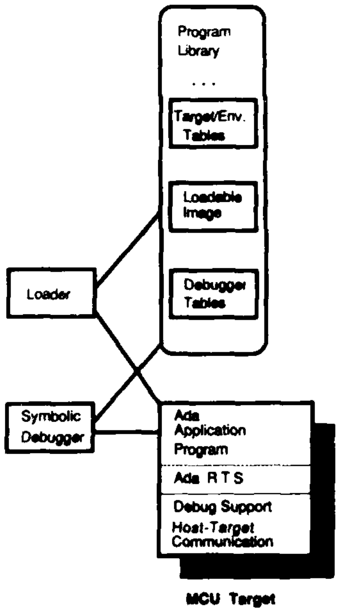


Fig. 4b: Ada cross development environment for *the HARM integration programme* (cont.)

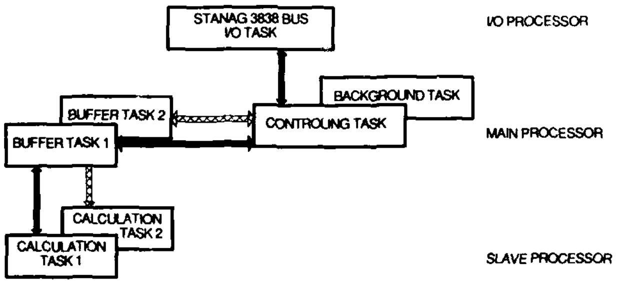


Fig. 5: *Ada software structure for a multiprocessor target*

DISCUSSION

M.Pitarys, US

1. What was your requirement for the time it takes to perform an Ada rendezvous?
2. Do you implement a shareable run-time system?

Author's Reply

1. 500 microseconds
2. No

LIST OF ATTENDEES

APPRIOU, A. Mr	ONERA, 29 Av. de la Division Leclerc, BP 72, 92322 Châtillon, France
ARDKEY, R.S., II Mr	Computer Technology Associates, Inc., 900 Heritage Drive, Ridgecrest, CA 93555, USA
BAKKEN, P.M. Dr	Director of Research, ELAB, 7034 Trondheim — NTH, Norway
BARAY, M. Prof.	Computer Science Dept., Bilkent Univ., P.K. Maltepe, Ankara, Turkey
BARET, H.D.S. Cdt.	EM FAe-VDT/B, 1 rue d'Evere, 1140 Bruxelles, Belgium
BART, J.J. Mr	Technical Director, Rome Air Development Center, Griffiss AFB, N.Y. 13441-5700, USA
BASSO. E. Mr	Agusta Sistemi, Spavia Isonzo 33, Tradate (Varese), Italy
BATZ, J. Mr	ADA Joint Program Office, Rm 3E114, The Pentagon, Washington D.C. 20301, USA
BECHER, P. Dr	MBB GmbH, FE 301, P.O. Box 80 11 60, 8000 München 80, FRG
BENNER, K.M. Lt	RADC/COES, Griffiss Air Force Base, NY, 13441-5700, USA
BERGGREN, C. Ms	IBM, MS0906, Federal Systems Division, Owego, NY 13827, USA
BERTRAND, C. Mrs	S.A.T., G.BOA22, 41 rue Cantagrel, 75013 Paris, France
BLAYLOCK, J.G. Mr	General Dynamics Corp., Ft Worth Division, 7229 Francisco Dr, Ft Worth, Texas 76108, USA
BLIN, M. Mr	Aérospatiale, 48 Route de Verneuil, 78130 Les Mureaux, France
BLOOMFIELD, A.P.D. Mr	Radar Systems Division, Ferranti, Crewe Toll, Ferry Road, Edinburgh, UK
BOARDMAN, R.M. Mr	Marconi Software Systems, Elstree Way, Borehamwood, Herts, UK
BOCKING, K. Mr	Carl Zeiss, SO.E1, Postfach, 7082 Oberkochen, FRG
BREVOT, J.G. Col.	BPM/NAC, 24 Bld Victor, 75996 Paris Armées, France
BRICE J.M. Mr	Directeur Technique, B.P.123, 38521 Saint Egrève Cedex, France
BRINK, J.L. Ms	Control Data Corp., 8800 Queen Avenue S., PO Box 1305, BLCS2T, Bloomington, MN 55431, USA
BUSSER, R.D. Mr	Allied/Signal Aerospace, Bendix/King Air Transport Avionics Division, 2100 NW 62nd Street, Fort Lauderdale, FL 33310, USA
BUSSIENNE, M. Mr	S.A.G.E.M., Chaussée Jules César, 95520 Osny, France
BUSTAMANTE, J.L. Mr	Avions Marcel Dassault, Breguet Aviation, 92214 St Cloud, France
CHARVOZ, J-R. Mr	Software Engineering Group Manager, Thomson-CSF Division, 178 Bd Gabriel Péri, 92240 Malakoff, France
CHONG HOK YUEN, C.K.S. Dr	Computer Science Department, College Militaire Royal St Jean, St Jean, J0J 1R0, Quebec, Canada
CHRISSOCHOIDIS, A. Col.	HAFGS, Branch C'/C/3 T.G.A. 1010, Holargos, Athens, Greece
CHRISTENSEN, E.B. Ms	Project Manager, DDC International A/S, DK 2300 Lyngby, Denmark
COLMANT, O. Capt.	EMFT GSP, Quartier Reine Elisabeth, 1 rue d'Evere, 1140 Bruxelles, Belgium
CONVERSE, R.A. Mr	Operations Manager, Systems Division, Computer Sciences Corp., Falls Church, VA 22046, USA
COOPER, J. Mr	President, Anchor Software Management, 5109 Leesburg Pike, Suite 214, Falls Church, VA 22041, USA
CORBISIER, F. Col.	HQ AAFCE (BE) PART, Ramstein Air Base M3, FRG
CROSETTI, G.M. Dr	Aeritalia, Gruppo Sistemi ed Equipaggiamenti, 10072 Caselle (Torino), Italy

CROVELLA, L. Dr	Aeritalia, Gruppo Sistemi Avionici ed Equipaggiamenti, 10072 Caselle (Torino), Italy
CURTIS, J.E. Mr	Ferranti, Defence Systems Ltd, Radar Systems Dept., Ferry Road, Edinburgh EH5 2XS, UK
DANSAC, J. Mr	Thomson CSF, Division Avionique, Générale, 52 Rue Guynemer, 92132 Issy-les-Moulineaux, France
DE THOMAS, A.P. Dr	AFWAL/FIGX, Wright Patterson AFB, OH 45433-6553, USA
DELOBEL, M.P. Capt.	Etat Major Force Aérienne (VDT/B), Quartier Reine Elisabeth, B-1140 Bruxelles, Belgium
DICKERSON, M.C. Major	6510 TW/TEVF, Edwards Air Force Base, CA 93523-5000, USA
DOREY, J. Mr	ONERA, 29 Avenue de la Division Leclerc, 92320 Châtillon, France
DUNCAN, G.W.L. Dr	Ferranti Defence Systems Ltd, Silverknowes, Ferry Road, Edinburgh EH4 4AD, UK
EPPINGER, D. Mr	Tengstr. 38, 8070 Ingolstadt, FRG
ESPOSITO, D. Lt. Col.	DASRS UFF. PD, Aeroporto, 00040 Pratica di Mare, Italy
FAGES, F. Mr	Thomson CSF LCR, Domaine de Corbeville, BP 10, 91401 Orsay, France
FLEMMING, D. Mr	MBB Dynamics Division, PO Box 80 11 49, 8000 München 80, FRG
FOURNET, P. Mr	Thomson/CSF, RCR, 178 Bld Gabriel Péri, 92242 Malakoff, France
GAGER, D.R. Mr	GEC Sensors, Christopher Martin Road, Basildon, Essex SS14 3EL, UK
GAGGIN, D.V. Mr	Director AVRADA, US Army Avionics R&D Activity, SAVAA-D, Fort Monmouth, N.J. 07703-5401, USA
GARTNER, O.H. Mr	Principal Planning Eng., SHAPE Technical Centre, P.O. Box 174, The Hague, Netherlands
GERMAIN, F. Maj.	TDLM/CT Kwartier Mousian, 3-1801 Peutie, Belgium
GRAEVENITZ, V. LTC.	BMVG — FUL VI 4, Postfach 1328, 5300 Bonn, 1, FRG
GUILLAUME, J.M. Mr	SFENA, Boite Postale No.59, 78141 Vélizy-Villacoublay, France
GUIOT, R. Mr	78, Quai Marcel Dassault, 92214 Saint Cloud Cedex, France
GURAN, H. Dr	Tubitak Ankara, ODTU — Balgat/Ankara, Turkey
HALL, C.W. Mr	Code 3922, Naval Weapons Center, China Lake, CA 93555, USA
HERRMANN, I.G.K. Maj.	BMVG Air Staff, P.O. Box 1328 Room 615, 5300 Bonn 1, FRG
HOIVIK, L. Dr	Informasjonskontroll ALS, P.O. Box 265, N-1371 Asker, Norway
HOLLEBEEK, J.J. Mr	Aeronautical Inspection Directorate, P.O. Box 7555, 1117 ZU Schiphol, Netherlands
HOLZBAUR, U.D. Dr	AEG Radar Subdivision, Dev., Data, Processing, Avionics Radar Syst Gp, Sedanstrasse 10, D-7900 Ulm, FRG
HOWELL, P.B. Mr	Manager, Advanced Transport, Operating Systems Program Office, Langley Research Center, Mail Stop 265, Hampton, VA 23665, USA
HUNT, G.H. Dr	ADXRE, Royal Aircraft Establishment, Farnborough, Hants GU14 6TD, UK
JACKSON, F.W. Mr	Marconi Space Systems, Anchorage Road, Portsmouth, Hants PO3 5PU, UK
JACOBSEN, M. Mr	AEG Aktiengesellschaft A14 V3, Sedanstr. 10, 7900 Ulm, FRG
JANSEN, B. Mr	c/o A.J. van de Goor, Delft University of Technology, Faculty of Electrical Engineering, Section of Computer Architecture, Mekelweg 4, 2628 CD Delft, Netherlands
JEAN, M. Dr	Computer Science Department, College Militaire Royal, Richelain, J0J 1R0 Quebec, Canada
JONES, L.G. Mr	STC, PO Box 174, 2501 The Hague, Netherlands
JOURNEAU, R. Mr	Thomson-CSF, DTG, 66 rue du Fosse Blanc, GX 1STT, B.P. 156, 92231 Gennevilliers Cedex, France

KAYA, D. Col.	Research and Development Dept, ARGE Ministry of Defence, Ankara, Turkey
KESICI, B. 1st Lt	1 H.I.B.M., K1 LG1 FB. MD, 26320 Eskişehir, Turkey
KEYDEL, W. Dr	DFVLR, Inst. for Radiofrequency Technology, Oberpfaffenhofen, D-8031 Wessling, FRG
KLEMM, R. Dr	FGAN-FFM, Neuenahrerstr. 20, D-5307 Wachtberg 7, FRG
KLENK, H. Dr	MBB GmbH, Abt, LKE 312, Postfach 80 11 60, D-8000 München 80, FRG
KROGER, A. Dipl. Ing.	MBB, TE 231, P.O. Box 95 01 09, D-2103 Hamburg 95, FRG
KRUEGER, C.M., Jr Dr	AFWAL/AS, Wright Patterson AFB, OH 45433, USA
KUHL, P. Mr	Dornier GmbH, Department CD, Postfach 1420, 7990 Friedrichshafen, FRG
KUTCHMA, E.K. Dr	Naval Weapons Center (Code 35), China Lake, CA 93555, USA
LACROIX, J.P. Mr	Thomson CSF AVG, 31 rue Camille Desmoulins, 92130 Issy les Moulineaux, France
LANGELLA, F. Maj.	DASRS-CGS, Aeroporto, 00040 Pratica di Mare (Roma), Italy
LAPORTE, C. Prof.	College Militaire St Jean, Dept de Physique, St Jean, P.Q. J0J 1R0, Canada
LASAXON V.M. Mr	c/o Computer Sciences Corp., 6510TW/TEVF, Edwards Air Force Base, CA 93523-5000, USA
LE GALL, P. Mr	Thomson Sintra — ASM, 94 Arcueil, France
LEMOINE, M. Mr	ONERA CERT DERF, 2 Av. E. Belin, 31055 Toulouse, France
LINDER, G. Mr	BWB FE VI 2 Landsberg, Iglingerstr. 280, D-8910 Landsberg, FRG
LONG, R. Capt.	US Air Force, AFWAL/AAAF-2, Wright-Patterson AFB, OH 45433-6543, USA
LORIAUX, G. Lt. Col.	SCRT ITA, Rue de la Fusée, 1130 Bruxelles, Belgium
LUSCHNITZ, W. Dipl. Ing.	AEG Aktiengesellschaft, Sedanstr. 10, D-7900 Ulm/Donau, FRG
MACPHERSON, R.W. Dr	DS POL 3/CRAD, 19NT, NDHQ, 101 Colonel By Drive, Ottawa, K1A 0K2, Canada
MAILLARD, J. Mr	Intertechnique, B.P. N.1, 78374 Plaisir Cedex, France
MANSEL, W. Mr	MBB GmbH, Military Aircraft Division, Postfach 80 11 60, D-8000 München 80, FRG
MARMELSTEIN, R.E. Lt	AFWAL/AAAF 2, Wright Patterson AFB, OH 45433-6543,USA
MASTELLONE, M. Eng.	Aeritalia SIAN, Viale dell'Aeronautica, 80038 Pomigliano D'Arco, Italy
MATTISSEK, A. Dr	LITEF GmbH, Abt. EWS, Postfach 774, 7800 Freiburg, FRG
MUENIER, M. Mr	Electronique Serge Dassault, 55 Quai Marcel Dassault, 92214 Saint Cloud, France
OZKAN, Y. Mr	3 Haura Ikmal Bakım RK Fabnke Midurligi, Etimesgut/Ankara, Turkey
PAMPOUCAS, Prof. Dr	Hellenic Air Force Academy, 22 Politeias Avenue, GR-14563, Greece
PARIS, G. Mr	Crouzet Aerospace, 25 Rue Jules Vedrines, 26027 Valence Cedex, France
PERCHANT, A. Mr	Aérospatiale, Dept E 54, 2 rue Beranger, 92322 Châtillon sous Bagneux, France
PITARYS, M.J. 1Lt	US Air Force, AFWAL/AAF 3, Wright Patterson Air Force Base, OH 45433-6543, USA
POTIER, D. Mr	Thomson CSF Division RCM, 178 Bd Gabriel Péri, 92240 Malakoff, France
RIESENER, J. Mr	WTD61-433, Flugplatz, D-8072 Manching, FRG
ROHE, I.K. Mr	IABG/WT-I, Einsteinstrasse 20, D-8012 Ottobrunn, FRG
RONTANI, B. Mr	Aérospatiale, Div. Hélicoptères, Département Equipements & Systèmes, B.P.13, 13725 Marignane, France
ROSE, K. Mr	N.D.R.E., P.O. Box 25, N-2007 Kjeller, Norway
SAMSON, P. ICA	STTE/AVI, 129 Rue de la Convention, 75731 Paris Cedex 15, France
SARAYDIN, A. Mr	Genelkermay J-6, Ankara, Turkey
SCHANG, T. Mr	Thomson CSF Division RCM, 178 Bd Gabriel Péri, 92240 Malakoff, France
SEALS, J.D. Dr	A&T Bell Laboratories, Whippany, NJ 07981, USA

SEPP, H. Dr	MBB GmbH, Space Communic. & Prop. Systems Div., KT1 Postfach 801169, 8000 München 80, FRG
SEVILLA E. Mr	Division de Simulacon y Avonia, C/LA Granja 44, Ceselsa, Poligono Industrial de Alcobendas, (Madrid), Spain
SILVA A. Ing.	Agusta Sistemi S.P.A., Via Isonzo 33, 21049 Tradate, Italy
SJOGREN J.A. Dr	USAAVRADA/AVSCOM, NASA Langley Research Center, Hampton, VA 23665-5225, USA
SPALDING, I.N. Mr	Smiths Industries, Bishops Cleeve, Cheltenham, Glos, UK
STOLL, M. Mr	AMD-BA, 78, Quai Marcel Dassault, 92214 Saint Cloud, France
SWIHART, D. Mr	Aeronautical Systems Division, F-16 System Program Office, Wright Patterson Air Force Base, OH 45433-6503, USA
TILLMAN, P.R. Mr	MBCS MCSSA, Software Sciences Ltd, Farnborough, Hants GU14 7NB, UK
TIMMERS, H.A.T. Ir.	National Aerospace Lab, NLR, Anth. Fokkerweg 2, 1059 CM Amsterdam, Netherlands
URSCHEL, W. Mr	Aeronautical Systems Division, F-16, System Program Office, Wright Patterson AFB, OH 45433-6503, USA
USGURLU, V. Mr	2nci IBHM C.lığı, FB. MD. lüğü, Kayseri, Turkey
VAN DE GOOR, A.J. Mr	Delft University of Technology, Faculty of Electrical Engineering, Section of Computer Architecture, Mekelweg 4, 2628 CD Delft, Netherlands
VAN OIJEN, A. Mr	OCE Nederland BV, R&D, 3606, P.O. Box 101, 5900 MA Venlo, Netherlands
VANDENBOSSCHE, P.J. Mr	Bell Telephone, Dept TT345, F.Wellesplein 1, B-2018 Antwerpen, Belgium
VELEZ, C.B. Dr	Computer Technology Associates, 5670 Greewood Plaza Blvd # 200, Englewood, CO 80111, USA
VILLEDIEU, P. Mr	Aérospatiale Division Hélicoptères, 13725 Marignane Cedex, France
VIVANCOS, A. Mr	Electronique Serge Dassault, 55 Quai Marcel Dassault, 92214 St Cloud, France
WAGNER, W. Dr	MBB GmbH, Hubschrauber und Flugzeuge, LKE 374, Postfach 80 11 60, 8000 München 80, FRG
WARD, A.O. Mr	British Aerospace, Warton PR4 1AX, UK
WHITE, P. Dr	Rolls-Royce PLC, PO Box 31, Derby DE2 8BJ, UK
WIEDENMANN, R. Dr	MBB, Postfach 80 11 60, 8000 München 80, FRG
WIRT, R.L. Mr	Dir. of Avionics Engrg, ASD/ENAC, Wright Patterson AFB, OH 45433-6503, USA
WOHLGEMUTH, T. Major	Wehrtechnische Dienstelle für Luftfahrzeuge, Flugplatz, 8072 Manching, FRG

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-CP-439	ISBN 92-835-0483-6	UNCLASSIFIED

5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France
6. Title	SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS
7. Presented at	the Avionics Panel Symposium held in Ceşme, Turkey, 25—29 April 1988.

8. Author(s)/Editor(s)	9. Date
Various	November 1988

10. Author's/Editor's Address	11. Pages
Various	420

12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.

13. Keywords/Descriptors

Software	Fault tolerance
Hardware	Avionics
Computer architectures	

14. Abstract

Software Engineering has evolved rapidly but the gap between demand and software output continues to grow. By the end of the decade research programmes in North America, Europe and Japan will begin to produce results in the areas of software tools, computer architecture, etc. The symposium considered how these advances might be applied to the avionics systems of the nineties and beyond and their impact on our aspirations in areas such as operation, fault tolerance etc. The software element of modern weapon systems continues to grow in size and complexity; offering major advantages but also potential risks.

AGARD Conference Proceedings No.439
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS
Published November 1988
420 pages

Software Engineering has evolved rapidly but the gap between demand and software output continues to grow. By the end of the decade research programmes in North America, Europe and Japan will begin to produce results in the areas of software tools, computer architecture, etc. The symposium considered how these advances might be applied to the avionics systems of the nineties and beyond and their impact on our aspirations in areas such as

P.T.O.

AGARD-CP-439

Software
Hardware
Computer architectures
Fault tolerance
Avionics

AGARD Conference Proceedings No.439
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS
Published November 1988
420 pages

Software Engineering has evolved rapidly but the gap between demand and software output continues to grow. By the end of the decade research programmes in North America, Europe and Japan will begin to produce results in the areas of software tools, computer architecture, etc. The symposium considered how these advances might be applied to the avionics systems of the nineties and beyond and their impact on our aspirations in areas such as

P.T.O.

AGARD-CP-439

Software
Hardware
Computer architectures
Fault tolerance
Avionics

AGARD Conference Proceedings No.439
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS
Published November 1988
420 pages

Software Engineering has evolved rapidly but the gap between demand and software output continues to grow. By the end of the decade research programmes in North America, Europe and Japan will begin to produce results in the areas of software tools, computer architecture, etc. The symposium considered how these advances might be applied to the avionics systems of the nineties and beyond and their impact on our aspirations in areas such as

P.T.O.

AGARD-CP-439

Software
Hardware
Computer architectures
Fault tolerance
Avionics

AGARD Conference Proceedings No.439
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE ENGINEERING AND ITS APPLICATION TO AVIONICS
Published November 1988
420 pages

Software Engineering has evolved rapidly but the gap between demand and software output continues to grow. By the end of the decade research programmes in North America, Europe and Japan will begin to produce results in the areas of software tools, computer architecture, etc. The symposium considered how these advances might be applied to the avionics systems of the nineties and beyond and their impact on our aspirations in areas such as

P.T.O.

AGARD-CP-439

Software
Hardware
Computer architectures
Fault tolerance
Avionics

operation, fault tolerance etc. The software element of modern weapon systems continues to grow in size and complexity; offering major advantages but also potential risks.

Papers presented at the Avionics Panel Symposium held in Ceşme, Turkey, 25—26 April 1988.

ISBN 92-835-0483-6

operation, fault tolerance etc. The software element of modern weapon systems continues to grow in size and complexity; offering major advantages but also potential risks.

Papers presented at the Avionics Panel Symposium held in Ceşme, Turkey, 25—26 April 1988.

ISBN 92-835-0483-6

operation, fault tolerance etc. The software element of modern weapon systems continues to grow in size and complexity; offering major advantages but also potential risks.

Papers presented at the Avionics Panel Symposium held in Ceşme, Turkey, 25—26 April 1988.

ISBN 92-835-0483-6

operation, fault tolerance etc. The software element of modern weapon systems continues to grow in size and complexity; offering major advantages but also potential risks.

Papers presented at the Avionics Panel Symposium held in Ceşme, Turkey, 25—26 April 1988.

ISBN 92-835-0483-6